

AD-A266 176



WL-TR-93- 8019

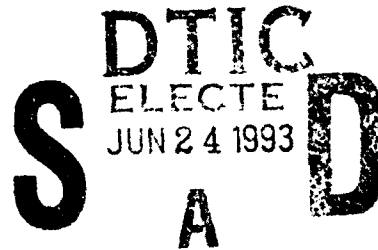


**INTEGRATION TOOLKIT AND METHODS (ITKM)
MANUFACTURING METHODS & PROTOTYPE TOOLKIT**

A Combined Modeling Method Using IDEF0, IDEF1x and XSpec

Industrial Technology Institute
P.O. Box 1485
Ann Arbor, MI 48106

July 1992



Final Report for Period June 1991 June 1992

Approved for public release; Distribution is unlimited.



Manufacturing Technology Directorate
Wright Laboratory
Air Force Systems Command
Wright-Patterson Air Force Base, Ohio 45433-7739

93-14186



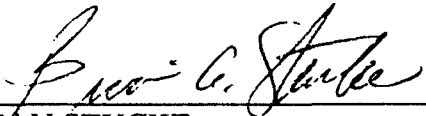
5280

NOTICE

When Government drawings, specifications, or other data are used for any purpose other than in connection with a definitely related Government procurement operation, the United States Government thereby incurs no responsibility nor any obligation whatsoever; and the fact that the government may have formulated, furnished, or in any way supplied the said drawings, specifications, or other data, is not to be regarded by implication or otherwise as in any manner licensing the holder or any other person or corporation, or conveying any rights or permission to manufacture, use, or sell any patented invention that may in any way be related thereto.

This report has been reviewed by the Office of Public Affairs (ASD/PA) and is releasable to the National Technical Information Service (NTIS). At NTIS, it will be available to the general public, including foreign nations.


This technical report has been reviewed and is approved for publication.



BRIAN STUCKE
Project Manager

5 Mar 93

DATE



BRUCE A. RASMUSSEN, Chief
Integration Technology Division
Manufacturing Technology Directorate

4 Mar 93

DATE

"If your address has changed, if you wish to be removed from our mailing list, or if the addressee is no longer employed by your organization please notify WL/MTIA, W-PAFB, OH 45433-6533 to help us maintain a current mailing list."

Copies of this report should not be returned unless return is required by security considerations, contractual obligations, or notice on a specific document.

REPORT DOCUMENTATION PAGE			FORM APPROVED OMB NO. 0704-0188	
Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing this burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.				
1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE July 1992		3. REPORT TYPE AND DATES COVERED Final Report June 1991 - June 1992
4. TITLE AND SUBTITLE Integration Toolkit and Methods (ITKM) Manufacturing Methods & Prototype Toolkit; A Combined Modeling Method Using IDEF0, IDEF1x and XSpec			5. FUNDING NUMBERS PE: 78011F C: F33615-91-C-5725 PR: 3095 TA: 06 WU: 64	
6. AUTHOR(S)				
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) Industrial Technology Institute P.O. Box 1485 Ann Arbor, MI 48106			8. PERFORMING ORGANIZATION REPORT NUMBER ITI-CR-92-57a	
9. SPONSORING MONITORING AGENCY NAME(S) AND ADDRESS(ES) Brian Stucke (513-255-7371) Manufacturing Technology Directorate (WL/MTIA), Bldg. 653 Wright Laboratory, 2977 P St., Suite 6 Air Force Systems Command, WPAFB OH 45433-7739			10. SPONSORING/MONITORING AGENCY REP NUMBER WL-TR-93- 8019	
11. SUPPLEMENTARY NOTES				
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for Public Release; Distribution is unlimited.			12b. DISTRIBUTION CODE	
13. ABSTRACT This report develops the major theoretical concepts for the integration of three diverse engineering modeling languages: IDEF0, IDEF1x, and XSpec. IDEF0 is a high level modeling tool based on functional decompositions, IDEF1x is a detailed data modeling tool and XSpec is a simulation tool based on object-oriented decompositions. The approach used to integrate the tools is to develop a meta-model of the three modeling languages. These meta-models are examined, and a single combined meta-model is developed. All three systems use the same combined meta-model. This ensures that any changes to the system made in one language is reflected in the other languages automatically. The report is divided into the following sections. A short description of each of the languages is given. These are not intended to be exhaustive, they are provided so that the reader is familiar with the terminology used in each of the methods and to get a feel of the types of models each language produces. The meta-models of IDEF0 and XSpec are given. The approach used to develop the combined meta-model is given. Finally, a complete description of the meta-model is presented.				
14. SUBJECT TERMS IDEF, IDEF0, IDEF1x, XSpec, Activity Based Modeling, Engineering Modeling Language, Object Oriented, Meta-Model.			15. NUMBER OF PAGES 52	
			16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT Unclassified	18. SECURITY CLASS OF THIS PAGE Unclassified	19. SECURITY CLASS OF ABSTRACT Unclassified	20. LIMITATION ABSTRACT SAR	

Standard Form 298 (Rev 2-89)
Prescribed by ANSI Std Z39-18
298-102

Table of Contents

1. Introduction	1
2. IDEF0	1
2.1 Activity Based Modeling	1
2.2 Decomposition	3
2.3 IDEF0 Method	5
3. IDEF1x	6
3.1 Entities	6
3.2 Attributes	8
3.3 Keys	8
3.4 Relations	8
3.5 Method	11
4. XSpec	12
4.1 Elements	12
4.2 Components	13
4.3 Messages, Pins, Connectors and Cables	14
4.3.1 Pins	15
4.3.2 Connectors	15
4.3.3 Cables	15
4.4 Method	16
5. Meta-Models of the Base Languages	17
5.1 A Meta-Model of a Hierarchical System	17
5.2 IDEF0 Meta-Model	18
5.2.1 IDEF0 Object Hierarchy	18
5.2.2 Concept Hierarchy	21
5.2.3 Links and Paths	21
5.2.4 Path	22
5.3 XSpec Model	23
5.3.1 Object Hierarchy	23
5.3.2 Terminal Hierarchy	23
5.3.3 Paths	26
5.3.4 Cable Hierarchy	26
6. The Combined Model	28
6.1 Relation Between IDEF0 Model and an XSpec Model	28
6.1.1 XSpec and IDEF0 Externals	28
6.1.2 Base Mechanisms	29
6.1.3 Foreign Flows and Transactions	30
6.1.4 Consistency	31
6.1.5 Remarks	31

6.2	Relation Between IDEF1x and XSpec Models	32
6.3	Relation Between IDEF1x and IDEF0 Models	33
6.4	Restrictions Placed on Models	33
6.5	The Modeling Process Using the Combined Toolkit	34
6.6	The Combined Meta-Model	38
6.6.1	Base Mechanisms and Externals	38
6.6.2	External concepts	40
6.6.3	External Messages	41
6.6.4	Transactions	41
6.6.5	Foreign Path	41
6.7	Generating a Prototype XSpec Model	44
6.7.1	Defaults	44
6.7.2	Generation Algorithm	45

List of Figures

Figure 1: An IDEF0 Activity	2
Figure 2: A Simple IDEF0 Diagram of a Manufacturing System	3
Figure 3: The Tree Representing the Activity Decomposition of an IDEF0 Model	4
Figure 4: Concept Joins and Splits	5
Figure 5: IDEF1x Entities	7
Figure 6: A Simple IDEF1x Model of a Student Record System	10
Figure 7: A Component Diagram	13
Figure 8: An IDEF1x Model of a Hierarchical System	18
Figure 9: The IDEF0 Object Model	20
Figure 10: The Concept Hierarchal System	21
Figure 11: IDEF0 Links and Paths	22
Figure 12: THE XSpec Object Model	24
Figure 13: The Terminal Structure	25
Figure 14: The Cable Hierarchy	27
Figure 15: Example of Disjoint Nodes Which Cover a Tree	29
Figure 16: The Combined Toolkit	37
Figure 17: Base Mechanisms and Externals	39
Figure 18: Relation of Messages to Concepts	40
Figure 19: Relation of Paths and Wires	42
Figure 20: Combined IDEF0 and XSpec Meta-Model	43
Figure 21: Prototype Model Generator	45
Figure 22: Recursive XSpec Diagram Generator	46

Accession For	
NTIS	<input checked="" type="checkbox"/>
DTIC	<input type="checkbox"/>
Unpublished	<input type="checkbox"/>
Justification	
By	
Date	
Availability Codes	
Dist	Special
A-1	

A Combined Modeling Method Using IDEF0, IDEF1x and XSpec

1. Introduction

This report develops the major theoretical concepts for the integration of three diverse engineering modeling languages: IDEF0, IDEF1x, and XSpec. IDEF0 is a high level modeling tool based on functional decompositions, IDEF1x is a detailed data modeling tool and XSpec is a simulation tool based on object-oriented decompositions.

The approach used to integrate the tools is to develop a meta-model of each of the three modeling languages. These meta-models are examined, and a single combined meta-model is developed. All three systems use the same combined meta-model. This ensures that any changes to the system made in one language is reflected in the other languages automatically.

The report is divided into the following sections. A short description of each of the languages is given. These are not intended to be exhaustive, they are provided so that the reader is familiar with the terminology used in each of the methods and to get a feel of the types of models each language produces. The meta-models of IDEF-0 and XSPEC are given. The approach used to develop the combined meta-model is given. Finally, a complete description of the meta-model is presented.

2. IDEF0

The Air Force Integrated Computer Aided Manufacturing (ICAM) program has developed several standards for describing systems. These standards are collectively called the ICAM definitions (IDEF). One of these standards is IDEF0 which describes the functionality of a system or an organization. IDEF0 incorporates the principles of abstraction and modularity in a graphical language.

2.1 Activity Based Modeling

The basic modeling construct in IDEF0 is an activity. An activity denotes the functions performed by a system. An activity transforms a set of inputs. Unlike other function-oriented modeling languages, IDEF0 classifies the inputs to an activity into three categories: inputs, controls and mechanisms. An input is something that is transformed by the function. A control determines when the function is to be performed or it can constrain the function in some manner. Finally, a mechanism is a resource that is required by the activity to perform the function. Notice that inputs are consumed by the function, while controls and mechanisms are not.

An activity is represented by a labeled box. Unlike most graphical languages, each side of a box has meaning. The left side is used for the inputs to the activity, the top is used for the controls, the right side is used for the outputs and the bottom is used for the mechanisms. Inputs, controls, outputs and mechanisms are collectively called either ICOMs or concepts.

Consider the situation where a person is to inspect parts coming off various machines. This situation is easily modeled by an IDEF0 activity. The incoming parts are the inputs to this function and the inspected parts are the outputs. If a foreman determines the order that the parts are to be inspected, then these decisions are the controls on the function. The design values on the blue print are the constraints on the inspection process, and therefore, the blue prints are also modeled as a control on the activity. The mechanisms are the inspector and the gauges he needs to perform the inspection process. The graphical representation of this activity is shown in Figure 1.

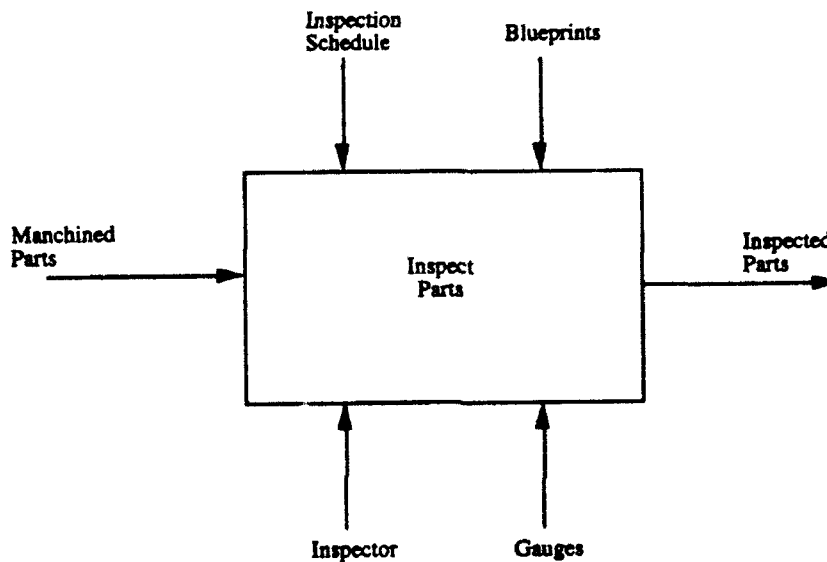


Figure 1: An IDEF0 Activity

A complete IDEF0 diagram of a simple manufacturing process is shown in Figure 2. Arrows are used to denote the connection of the outputs of one activity to the inputs, controls or mechanisms of other activities. Each arrow is labeled with the concept that flows along it.

Activities have dominance. When constructing IDEF0 diagrams, it is customary to put the activities that occur first in the upper left corner of the diagram. Activities which occur later are placed lower and to the right. Some ICOMS do not terminate on an activity box. These ICOMS represent flows into and out of the system. These flows are called externals.

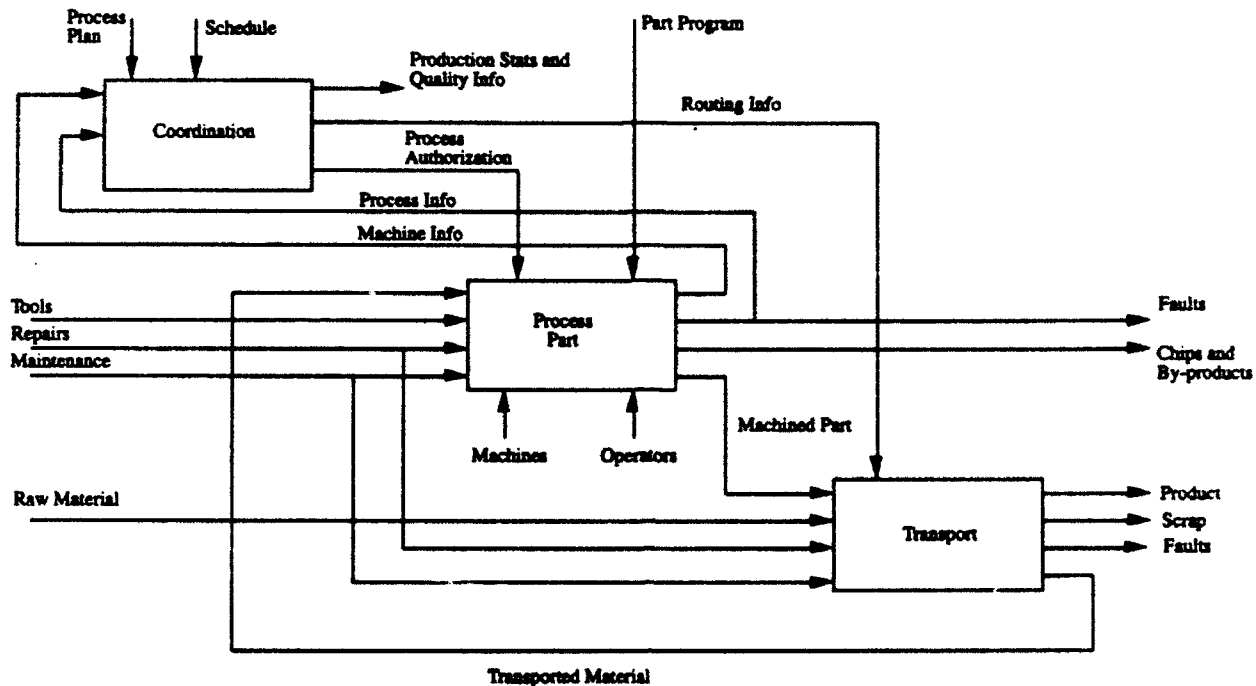


Figure 2: A Simple IDEF0 Diagram of a Manufacturing System

2.2 Decomposition

A primary feature of IDEF0 models is decomposition. Any activity in an IDEF0 model can be decomposed into another entire IDEF0 diagram sometimes referred to as the child diagram. The concepts of the parent activity will serve as sources and sinks of concepts in the child diagram. Although these look like externals, they are not since flows terminate on some other diagram in the model. If an external occurs in a diagram, it is denoted by placing a set of parentheses around the terminal end of the flow. An activity and its child diagram must be "plug-for-plug compatible", that is, for each ICOM on the parent activity there must be a corresponding sink or source in the child diagram.

The decomposition of activities is recursive. It is standard practice that the top level IDEF0 diagram only contains a single activity. This activity denotes the entire system. The arrows then represent all of the flows which cross the boundary of the system. Such a diagram is called the context diagram.

Activity decompositions are easily represented by a tree. Each node in the tree is an activity in the IDEF0 model. The root of the tree is the single activity in the context diagram. The nodes attached to the root of the tree are the activities in the child of the context diagram, and so forth. An example of this tree is shown in Figure 3. The activities at the leaves of the tree are called the base activities. The collection of all the base activities are all the functions that are actually performed by the system. The other activities are artifacts of the modeling process. They represent logical collections of base activities and act as an aid in understanding the system. A model whose decomposition can be represented by tree is called a hierarchal model.

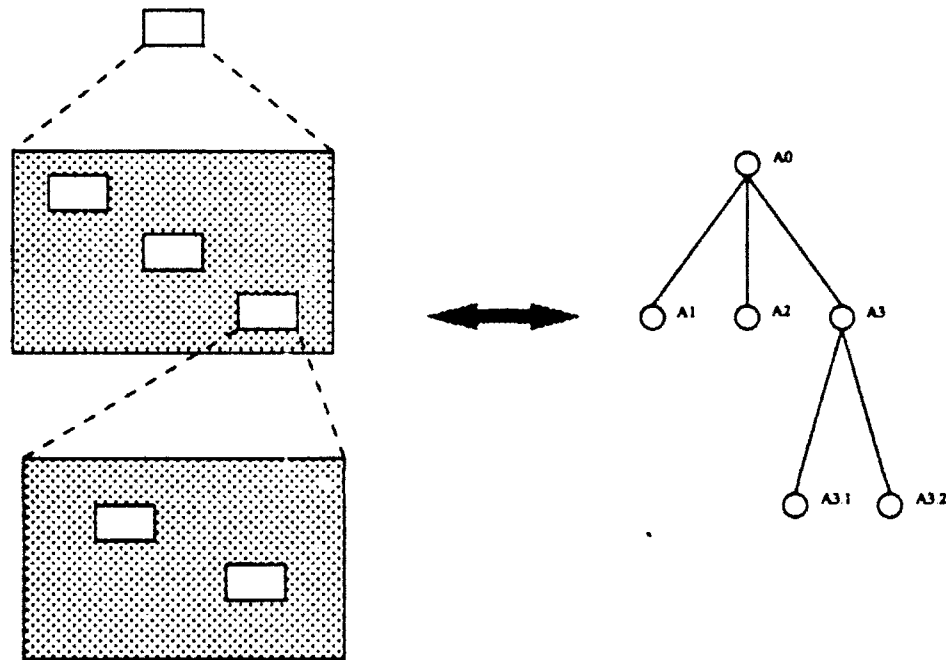


Figure 3: The Tree Representing the Activity Decomposition of an IDEF0 Model

In IDEF0 concepts are decomposed just like activities. For example the concept "parts" may be composed of the concept "good parts", "bad parts", "machined parts" and "inspected parts". "Machined parts" can be further decomposed into other concepts. the decomposition of concepts is shown graphically on an IDEF0 diagram by arrows which are split or joined. For example, suppose the concept A is composed of the concepts B and C. Algebraically, this can be denoted as

$$A = B + C.$$

If *A* is the output of one activity and *B* and *C* are inputs of two different activities, then the IDEF0 diagram would show a split of the arrow corresponding to the *A* concept (see the partial IDEF0 diagram in Figure 4). Also notice how concepts *X*, *Y* and *Z* are joined together to form the single concept *M* which is the input of some activity in the model.

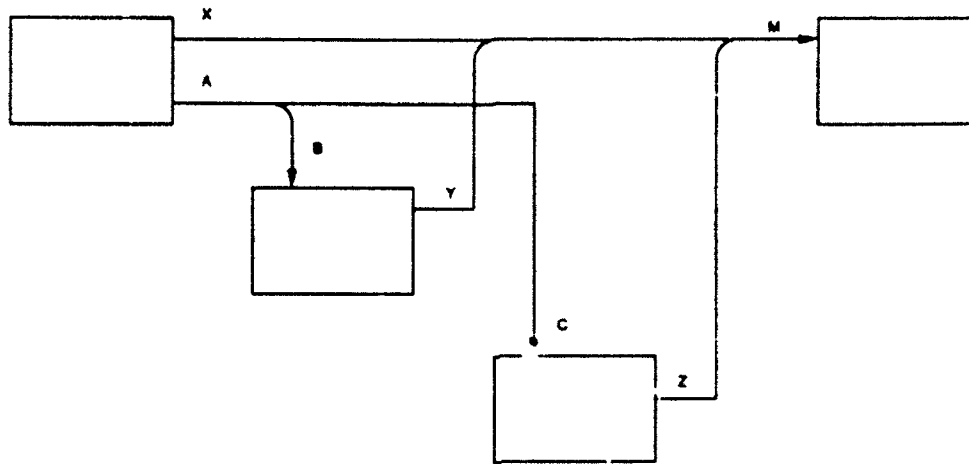


Figure 4: Concept Joins and Splits

There are many minor details about the syntax of IDEF0 models that has not been described here. These are mainly concerned with drawing and labeling conventions. Since these conventions are not important to the main thrust of this work, they will not be discussed.

2.3 IDEF0 Method

This section describes the typical IDEF0 modeling process. This description focuses on just the creation of the IDEF0 model and not on the other activities involved in a larger modeling effort. The modeling process consists of the following major steps.

1. **Gather Information.** The first step in creating an IDEF0 model is to gather information about the system to be modeled. Information sources usually include: documents about the system, interviews with experts, surveys of the users of the system, observation of the existing operations, and information from previous related projects.
2. **Determine the Purpose and Viewpoint.** IDEF0 models have a specific purpose and viewpoint. The purpose is defined as stating the question that the model must answer and the viewpoint is the perspective from which the model is to be described. The analyst should document these early on in the modeling process.

3. **Create Data List.** From all the information that was gathered, all the potential concepts are listed. Concepts are things that are used during the operation of the system. Concepts are grouped into aggregates. That is, the concepts which seem to belong to the same categories are grouped together.
4. **Create Activity List.** From all the information that was gathered, the functions or activities that are performed by the system are listed. The categories of data that are used by each function are then listed. This may help in further aggregating the data lists. Finally, the activities are clustered to create an initial activity hierarchy.
5. **Define Context.** The context diagram is drawn. This will clearly indicate the scope of the system and the concepts which are the interface to the external world.
6. **Decompose the Activities.** The subject of an activity to be decomposed is considered. The analyst determines how the activity performs its function within the purpose and viewpoint of the model. A trial set of activities is drawn for the child diagram. The arrows that touched the boundary of the parent activity are connected in the child diagram. A data list for the child diagram is created. The activities in the child diagram are connected and alternative decompositions are attempted.
7. **Review the model.** All the IDEF0 diagrams in the model are reviewed to check for consistency. Alternative decompositions and concept bundlings are tried. The model is checked to see if it meets the original stated purpose. All the data and activities that were initially listed are checked to see if they have been modeled. The model is checked for clarity and revised as appropriate.

3. IDEF1x

Another modeling standard that resulted from the ICAM effort is IDEF1x. IDEF1x is one of several related methods used to model the data used by an organization or a system. The primary purpose of the IDEF1x methodology is to model the conceptual schema of the data for the three schema database system as defined by the ANSI SPARC report on database design. Some tools can take an IDEF1x description and automatically generate the database schema designs.

3.1 Entities

The main modeling concept in IDEF1x is the data entity. A data entity represents a set of data instances. For example, the entity "Student" may keep the data about each student in a large university. It is important to realize that entities represent the data kept about an object, and they are not a representation of the physical object itself.

There are two types of entities: identifier-independent and identifier-dependent entities. An entity is identifier-independent if each data instance in the set represented by the entity can be uniquely identified without determining its relation with other entities in the model. An entity is identifier-dependent if unique identification of its data instances do depend on its relation to other entities. Identifier-independent entities can exist by themselves, while identifier-dependent entities are meaningless without their connection to other entities in the system. For example, the entity student is an a identifier-independent entity, each student can be identified by the student number or social security number; while an entity which represents an item in a purchase is meaningless without its connection to to the purchase order, and therefore, it is identifier-dependent.

Graphically, entities are denoted by boxes. If the box has square corners, then it represents an identifier-independent entity. Round corners are used to represent identifier-dependent entities. Each entity is assigned a unique name which is placed on top of the box. Figure 5 illustrates some examples of entities.

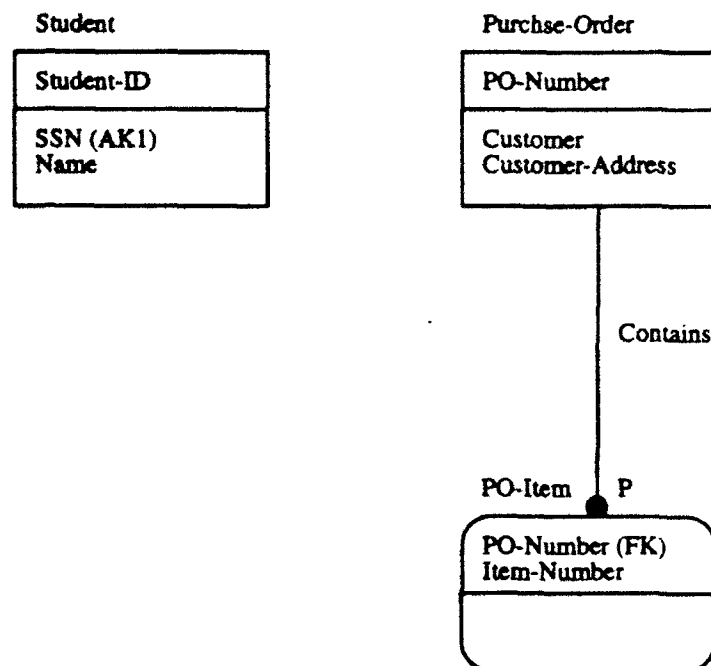


Figure 5: IDEF1x Entities

3.2 Attributes

Attributes represent a characteristic or property of an entity. An attribute instance is the value of the attribute for a single entity instance. Attributes have unique names.

An entity can own any number of attributes. However, an attribute can only be owned by a single entity. Attributes are shown on the model by listing their names, one per line, in the entity that they are associated with. In Figure 5 above, "Student-ID", "SSN", and "Name" are all attributes of the entity "Student".

Every instance of an entity must have a value for every attribute; this is called the "no null" rule. Also no instance of an entity can have more than one value for an attribute associated with that entity. This is called the "no repeat" rule.

3.3 Keys

Every entity must have an attribute or a combination of attributes that uniquely identifies each data instance for the entity. These attributes form the primary key for the entity. Primary keys are listed in the top section of the entity. They are separated from the rest of the attributes by a horizontal line. In Figure 5, Student-ID is the primary key for the entity "Student" and the combination of the attributes PO-Number and Item-Number form the primary key for the entity "PO-item".

In addition to attributes which are owned by an entity, an attribute can be inherited through a relation. In any IDEF1x model, a child entity can inherit any of the attributes that appear in the primary key of its parent entity. Inherited attributes are called foreign keys and are denoted by a FK in parenthesis after the attribute name. Foreign keys can be part of the primary key of the child entity. This is the case for the "PO-Item" entity in the above figure.

A single attribute may be inherited along two relations. In this case a role name can be prefixed to the attribute name. The role name is separated from the attribute name by a period. Role names are usually related to the relation that the attribute was inherited across.

Occasionally, there are several attributes (or combination of attributes) that can form the primary key for an entity. Only one of these attributes is assigned to be the primary key, the other attributes are called alternate keys. Alternate keys are denoted by an AK in parenthesis after the attribute name. If there is more than one alternate key for a given entity, then an integer suffix on the AK is used to uniquely identify each alternate key. For example, SSN is an alternate key for the "Student" entity.

3.4 Relations

Entities are related to one another. The relations are shown by lines connecting the entities in an IDEF1x diagram. A relation is an association between two entities in which each instance of one entity, called the parent entity, is related to zero, one or more instances of a second entity,

called the child entity. An example, is the relation between the "Purchase-Order" and the "PO-Item" entities in Figure 5. The child entity is denoted by a black dot placed at the end of the line representing the relation.

A solid line depicts an identifying relation between two entities. The child entity of an identifying relation is always an identifier- dependent entity. The primary key attributes of the parent inherited by the child must be part of the primary key of the child. The parent in an identifying relation must be an identifier-independent entity, unless it is a child of another identifying relation. A dashed line is used to denote a non-identifying relation. Both parent and child in a non-identifying relation will be identifier- independent entities, or children of some identifying relation.

Relationships are named. The names are placed next to the lines that denote the relation. Relationship names are verb phrases such that a sentence formed by combining the parent name, the relation name, and the child name expresses the intent of the relation. For example, The sentence "Purchase order contains PO items" clearly expresses the intent of the relation between the entities in Figure 5. The name of relations do not need to be unique within the IDEF1x model.

Each relation has a cardinality, which specifies how many child instances may be related to a single parent instance. IDEF1x supports four different cardinalities:

1. **Zero, One or More** Each instances of the parent entity is associated with zero, one or more instances of the child entity. This is denoted by just a dot at the child entity.
2. **One or More** Each instances of the parent entity is associated with one or more instances of the child entity. This is denoted by the letter P (for positive) near the dot at the child entity.
3. **Zero or One** Each instances of the parent entity is associated with zero or one instances of the child entity. This is denoted by the letter Z near the dot at the child entity.
4. **Exactly N** Each instances of the parent entity is associated with exactly N or more instances of the child entity. This is denoted by the numeral N near the dot at the child entity.

Occasionally, a relation between two entities cannot be represented in the simple parent child paradigm of IDEF1x. For example consider the relation between the entities student and class. Every student may attend several classes and classes will have many students in them. Neither of these can be a child entity, since both are related to more than one instance of the other entity. These are called non-specific relations, and are resolved by using an additional entity, whose instances represent a single pairing of the instances of the two entities. This situation is shown

in Figure 6. Each instance of the entity "Class-List" represents a class that a particular student is taking.

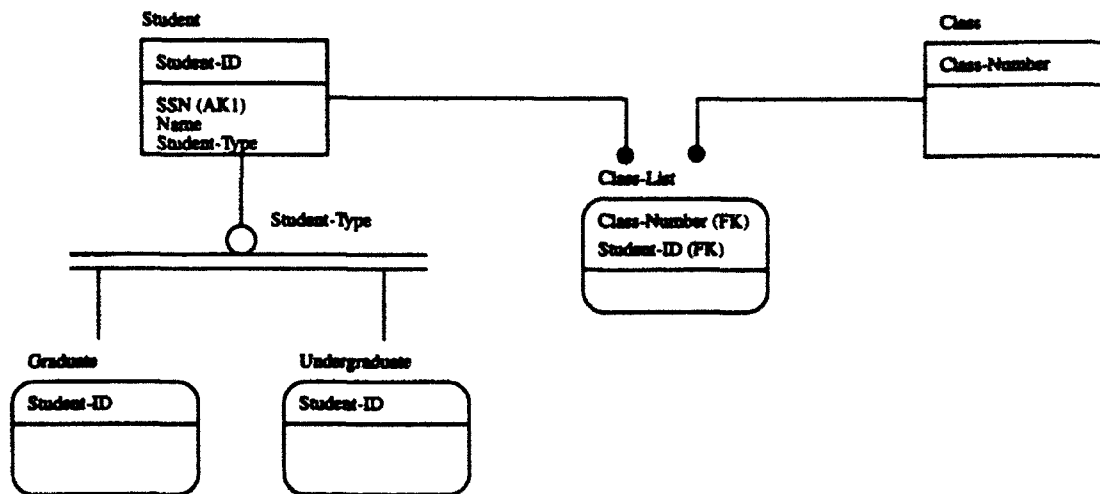


Figure 6: A Simple IDEF1x Model of a Student Record System

IDEF1x also support categorization relations. A complete categorization relationship is a relation between two or more entities in which one entity, called the generic entity, is related to an instance of one, and only one, of the other entities, called category entities. For example, student could be the generic entity and undergraduate and graduate could be the category entities. Therefore, in this model any student is either a graduate or an undergraduate and can never be both. An attribute in the generic entity instance determines which of the possible category entities it is related to. This attribute is called the discriminator. The category entities share all the attributes of the generic entity. In IDEF1x diagrams, however, only the primary key attributes of the generic entity are explicitly shown as being inherited by the category entities.

A categorization relation is denoted graphically by a circle and two lines. The circle is connected by a line to the generic entity, and the category entities are connected to the double lines. The name of the discriminator attribute appears in the generic entity and is also written next to the circle. Figure 6 shows a categorization relation.

A category relation can only have one generic entity. The category entities of one relation can be the generic entity of another relation. An entity may be the generic entity for any number of categorization relations. A category entity cannot be a child entity in a connection relation.

There is also an incomplete categorization relation. This relation is used in the early stages of IDEF1x models, it is like the complete categorization relation, except that it recognizes the

possibility that there may be more categories to be added later. All the rules mentioned above apply to both types of relations. Graphically, the only difference is that the incomplete categorization is drawn with a single line.

3.5 Method

The typical steps taken in the development of an IDEF1x model are described below.

1. **Collect Source Material.** As with any modeling method, the first step is the collection of the material needed to develop the model. Information sources usually include: documents about the system, interviews with experts, surveys of the users of the system, observation of the existing operations, and information from previous related projects. Many times an IDEF0 model is created first, and the concepts of this model are used to identify many of the entities and attributes of the data system.
2. **Identify Entities.** From the information gathered the entities are identified. Entities are things that can be described. There has to be several instances of an entity in the model and each entity must be uniquely identifiable. If a candidate entity describes something, it is usually an attribute of the thing (entity) that it describes. If an IDEF0 model is used as a source of material the concept list will contain potential entities. Remember that the IDEF0 model represents the flow of physical parts as well as data. The physical parts should not be considered as entities, but if data is associated with these parts, then entities should be defined to describe this data. Finally, names are assigned to the entities.
3. **Define Relations.** Identify the relations among the entities defined in the previous step. Some relations will be obvious. Other relations may be discovered by considering all the possible relations among the defined entities.
4. **Construct an Entity Diagram.** All the entities and relations determined above are placed on a diagram ignoring the keys and attributes. The parent, child and the cardinality of each relation is determined.
5. **Resolve Non-specific Relations.** Appropriate entities are added to resolve all the non-specific relations.
6. **Determine Key Attributes.** Using the attribute pool developed above, the key attributes are determined for each entity. Some entities will have several groups of attributes that can be a key for the entity. One of these must be selected as the primary key. The others are alternate keys. Some primary keys may be a result of inheritance, so the development of the primary keys of the identifier-independent entities is done first. All the primary keys are migrated along the relations to the child entities. Additional attributes are determined that

may be needed for the primary keys of the child entities. This process is continued until all the primary keys are determined.

7. **Identify Non-key Attributes.** The remaining attributes are added to the entities.
8. **Review and Refine Model.** The model is reviewed to see that all the data has been captured and that all the syntax rules of IDEF1x have been adhered to. Appropriate changes are made as required.

4. XSpec

XSpec is a graphical language developed by the Industrial Technology Institute (ITI) which is used to specify the interactions among the dynamic elements in a manufacturing system. The main strength of XSpec is that it specifies the structure of the system by decomposing it along physical boundaries. At the lowest level in the decomposition is an element. Further specification of each element is done in a language best suited for modeling the behavior of that element. Therefore, a manufacturing system is represented by many interconnected elements, modeled in diverse languages, with XSpec modeling the interactions.

XSpec system models provide enough detailed information that executable models can be created and simulated from the element specification. The specification is split across engineering disciplines, and encapsulated in the element structure. Executable code for the elements can be written on different simulation tools, each specializing in that discipline. The entire system is then simulated on an integrated tool such as XFaST which was developed by ITI. The dynamic performance of the system can be studied. This provides a powerful tool to design and analyze complex, interdisciplinary systems. Ultimately, the simulation source code becomes the specification for each element.

4.1 Elements

XSpec uses a construct called an *element* to represent, encapsulate and model the behavior of a portion of the system from the perspective of a single engineering discipline. Each element is a partial model created and executed on a tool designed for a particular engineering discipline. An element, like an object in object-oriented systems, includes data (state) and procedures (methods) which operate on that data. These models are built in such a way that they can be plugged together with other elements and executed on a computer.

Element models are written and executed on appropriate tools for their domain. These element models, which exist on diverse tools, are executed in a coordinated fashion. The tools are integrated into the XFaST framework which allows the individual elements to communicate with one another even if they are on different tools. Since XFaST is just a framework and not a design tool, it maximizes the leverage obtained by using existing design tools.

4.2 Components

In XSpec a *component* is a means to group the elements written in several diverse domain specific languages so they can be manipulated as a single entity. A component represents a cohesive functional unit of the system such as a servo mechanism, a robot, a machine, or even an entire subsystem like a drive train on an automobile.

Components and elements are connected together in a *component diagram*. These connections represent the interaction among the entities. Figure 7 depicts a typical component diagram. Components are represented by rectangles, elements by circles and the interactions by lines connecting the entities.

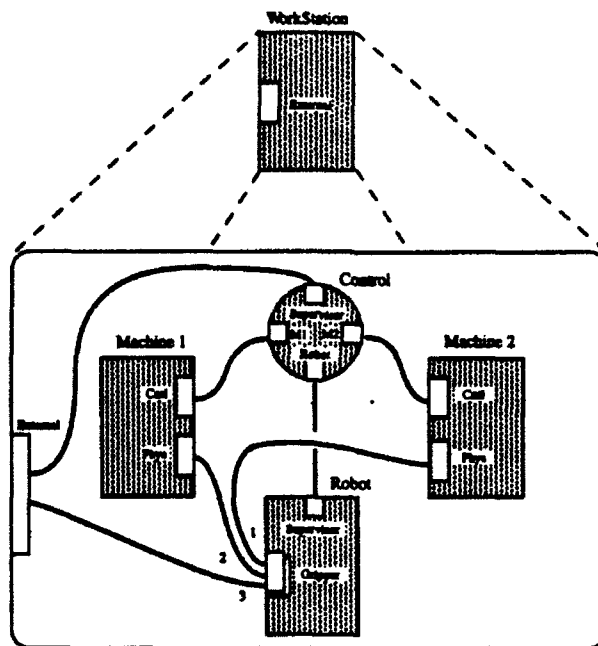


Figure 7: A Component Diagram

Component diagrams are organized in a series of levels each providing more detail about one of the components in the diagram above it. At the highest level is a *system context diagram*. This diagram consists of a single component which represents the model of the entire system connected to all its *external agents*.

A component serves as a structure to group related XSpec entities. Such a grouping operation allows a natural approach to system decomposition, provides a consistent notation for all disciplines, and enforces good design practice.

4.3 Messages, Pins, Connectors and Cables

XSpec uses the term *message* to describe all interactions between XSpec entities. An interaction is any flow of mass, energy or information between two parts of a system. The types of interactions are defined in the interface of each XSpec entity using *pins* that are grouped into *connectors*. A pin serves as the specification for an interaction (or message) between XSpec entities. It defines the type of the message and the types of the parameters it contains. The location of a pin identifies the source or destination of the message.

Messages are categorized into the following four classes:

1. **Event.** An event is a parameterless message which is used to model the instant that something occurs. Events are used primarily to model sensor firings and simple actuations.
2. **Signal.** A signal is a single parameter message. The parameter on a message is valid at all times. Signals are used to model discrete or continuously varying phenomena like temperature, position and velocities.
3. **Command.** A command is a multi-parameter message which is used to instruct an element to perform some function. Commands can be used to model both physical and data flows. The arrival of a part (the parameters describe the part), a move command to a robot from a controller, or a person entering data into the system can all be modeled by commands.
4. **Request.** A request is similar to a command, except the receiving element must eventually respond to the request. If a robot responds to the controller after each move command is finished executing, then the move should be modeled by a request, with the done indication being the response

Pins are strongly typed. Each pin specifies the message that flows across it, and the direction of the message (whether the message flows in or out of the element the pin is on). Messages can only flow between compatibly typed pins.

Messages flow along *cables* which connect pins among entities in a component diagram. A pin specifies the nature of an interaction between two entities, while a message represents a specific instance of that interaction.

4.3.1 Pins

A pin defines a single type of interaction that is available for an element or a component. Since all interactions occur through messages which are essentially data in the model, the interactions can be specified by typed data structures. Pin specifications are nothing more than the type definition of the message.

4.3.2 Connectors

Pins are grouped into *connectors*. The names of each connector on an entity must be unique. A connector provides a "window" or logical destination for messages. A message is sent to (or received from) a connector and pin pair. That is, the element sends a message by specifying the name of the pin and the name of the connector that the pin is on. The actual name of the connected element is not needed to either send or receive a message.

As components serve to group elements and components, so a *transaction* is used to group pins. Many times an interaction between two elements cannot be specified by a single message. For example, to model a power transfer between elements requires two signals, one for effort and the other for flow. It then makes sense to group both of these signals together in a single transaction to indicate that they both must be connected. Cables are a convenient tool that can be used to represent the flow of the entire transaction.

An *indexed connector* defines an array of like connectors that can be attached to multiple elements. An indexed connector is represented by a box drawn with a double line boundary. The double boundary is meant to imply the existence of multiple connectors, all of which have the same set of pins. Individual connectors in the group are referenced by the connector index.

4.3.3 Cables

When an XSpec component diagram is created, elements are created or retrieved from a library and their pins graphically connected together by *wires* and *cables*. Wires define how messages are routed among the elements by connecting pins on one element to the pins on another element. Cables bundle wires with common endpoints. Cables are generally not named. However, when several paths are connected to an indexed connector, each path is labeled with the index of the connector that it is attached to.

When a path is created, pins and subconnectors on one end of the path are automatically connected to pins and subconnectors at the other end with identical names, identical parameter types, and compatible pin types.

Cables, pins and messages form a three level abstraction mechanism for modeling interactions among elements. Cables indicate that an interaction exists between two elements without defining the details of that interaction. Component diagrams show the cables, so with a quick glance at the diagram designers can identify the interacting elements. Pins provide a detailed definition of the type of the interaction by defining the direction of information flow, the kind

the type of the parameters associated with the interaction. Finally, messages define the specific instances of each interaction along the time axis during the execution of a model. Separating these levels of detail aids in building the models, since many times the existence of an interaction, the details of an interaction type and the occurrence of the interaction are determined at different stages of model building.

4.4 Method

This section describes the typical XSpec modeling process. This process assumes that the XSpec model is created in isolation and is not part of a larger modeling effort. The modeling process consists of the following major steps.

1. **Gather Information.** The first step in creating an XSpec model is to gather information about the system to be modeled. Information sources usually include: documents about the system, interviews with experts, surveys of the users of the system, observation of the existing operations, and information from previous related projects.
2. **Create Initial Components.** From all the information that was gathered, the initial components are determined. These are usually identified by decomposing the system along the physical boundaries of the system. The components are decomposed into elements, with each element representing a particular engineering view of the component. Typical elements are the model of the physical aspects of the component and a model of the software needed to control the system.
3. **Define Context.** The context component diagram is drawn. This will clearly indicate the scope of the system. An initial component hierarchy is developed.
4. **List Connectors and Transactions.** From the functions that each element must perform, the types of transactions each element needs to perform is determined. A connector for each transaction on the elements is created.
5. **Create cables.** The elements are connected. The function of each element is reviewed and any additional transactions that are needed for the element to perform all of its requirements are added. Connectors are added to components if transactions flow across them. These component connectors are grouped based on common destinations. All the component diagrams are constructed.
6. **Decide on Pin Definition.** The specific messages required for each transaction is designed. The pins and wires for each of these messages is added along with a complete specification for each pin.
7. **Develop Elements.** Based on the required functionality of each element, a model of the interactions of all the pins on the element is developed. This is usually

done by modeling what happens when the element receives each of its input messages.

8. **Simulate the Model.** All the element specifications are transferred to executable simulation code, and the model is executed. The element definitions are refined based on the dynamic performance of the system.
9. **Review the model.** All the component diagrams in the model are reviewed and checked for consistency. Alternative component decompositions are tried. The model is checked to see if it meets the system requirements. The clarity of the model is checked and the model is revised as appropriate.

5. Meta-Models of the Base Languages

This section describes the meta-models for the IDEF0 and XSpec modeling languages. The purpose of these meta-models is to express the relations among the various data elements present in each of the modeling languages. The meta-models are expressed in IDEF1x notation. The ultimate goal is to describe a way to combine these individual meta-models into a single meta-model which represents the entire system.

5.1 A Meta-Model of a Hierarchical System

The IDEF0 and XSpec modeling paradigm have many hierarchical systems embedded in them. For example, activities and concepts form hierarchical systems in IDEF0 and components, connectors and cables are hierarchical systems in XSpec. So before proceeding with the description of the meta-models for IDEF0 and XSpec, a meta-model of a generic hierarchical system is described.

Every hierarchical system has a *root object*. The root object is decomposed into several other objects. This decomposition continues recursively until objects are reached that cannot be further decomposed. These are called the *base objects*. The IDEF1x model of such a structure is based on the following two observations:

1. Every object is either the root object or it is contained in some other object.
2. Every object either contains other objects or it is a base object.

An object that contains other objects is called a *composite object*.

These two observations suggest that a hierarchical system can be represented by a generic object entity which is related to four category entities through two classification relations (see Figure 8). To complete the data model, a non-identifying relation is used to identify the objects that are contained in a composite object.

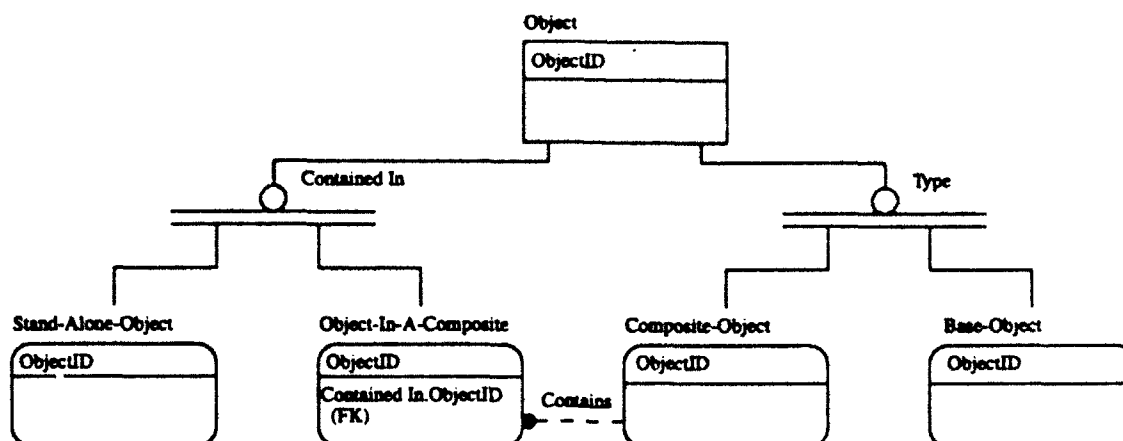


Figure 8: An IDEF1x Model of a Hierarchical System

5.2 IDEF0 Meta-Model

The IDEF0 meta-model is illustrated in Figures 9, 10 and 11. The entities in an IDEF model can be classified into three broad classes: objects, concepts and interaction among objects. Each of the entities and relations among the entities for the IDEF0 meta-model are explained in detail in this section.

5.2.1 IDEF0 Object Hierarchy

An IDEF0-Object is either the entire IDEF0 model, an activity, an external, or a place where concepts join or split. Figure 9 illustrates the categorization of an IDEF0-Object. The IDEF0-Object is first classified as either the entire model or a connectable object. An IDEF0-Connectable-Object is an object which is connected by concept flows to other connectable objects. The IDEF0-Model is a single entity that is used to represent the context diagram (the A-0 diagram) of the IDEF0 model. Clearly this object is not connected to other objects in the model. All IDEF0 objects have names and descriptions. These are recorded in the I-Description and I-Name attributes of the IDEF0-Object entity.

Connectable objects form a hierarchical system. However, a slight modification has been made in the standard hierarchal system meta-model developed in the previous section. Base objects can be one of three types:

1. Base-Activity - an activity which is not further decomposed.
2. IDEF0-External - a source or sink of concept flows which are external to the context of the model.
3. Join-Split-Merge-Spread - are points on an IDEF0 diagram at which concept flows either join, split, merge or spread..

IDEF0 syntax does not have graphical symbol for an external. So graphically an IDEF0-External is nothing more than a point in the IDEF diagram. We have included this as a separate entity to simplify the modeling of concept flows. Flows must always start and terminate on an IDEF0-Object. Like externals, Join-Split-Merge-Spread entities do not have a graphic symbol associated with them. They represent the actual point that a concept join, split, merge, or spread takes place in the diagram.

Notice that the IDEF0-External and Base-Activity entities are grouped together into a generic entity called IDEF0-Producer-Consumer. Both of these entities can be terminal points for an IDEF0-Path, (IDEF0-Paths are explained in detail later) and so it is convenient to create a generic entity for them.

Notice that there is a relation between the IDEF0-Model entity and the root entities in the connectable object hierarchy. Clearly, given the syntax of an IDEF0 model, if an IDEF0-Object is not in a connectable object, the only other place it can exist is in the context diagram, that is, the IDEF0-Model entity.

Finally, the attributes I-Size and I-Position in the entity IDEF0-Connectable-Object store the appropriate information to plot the objects.

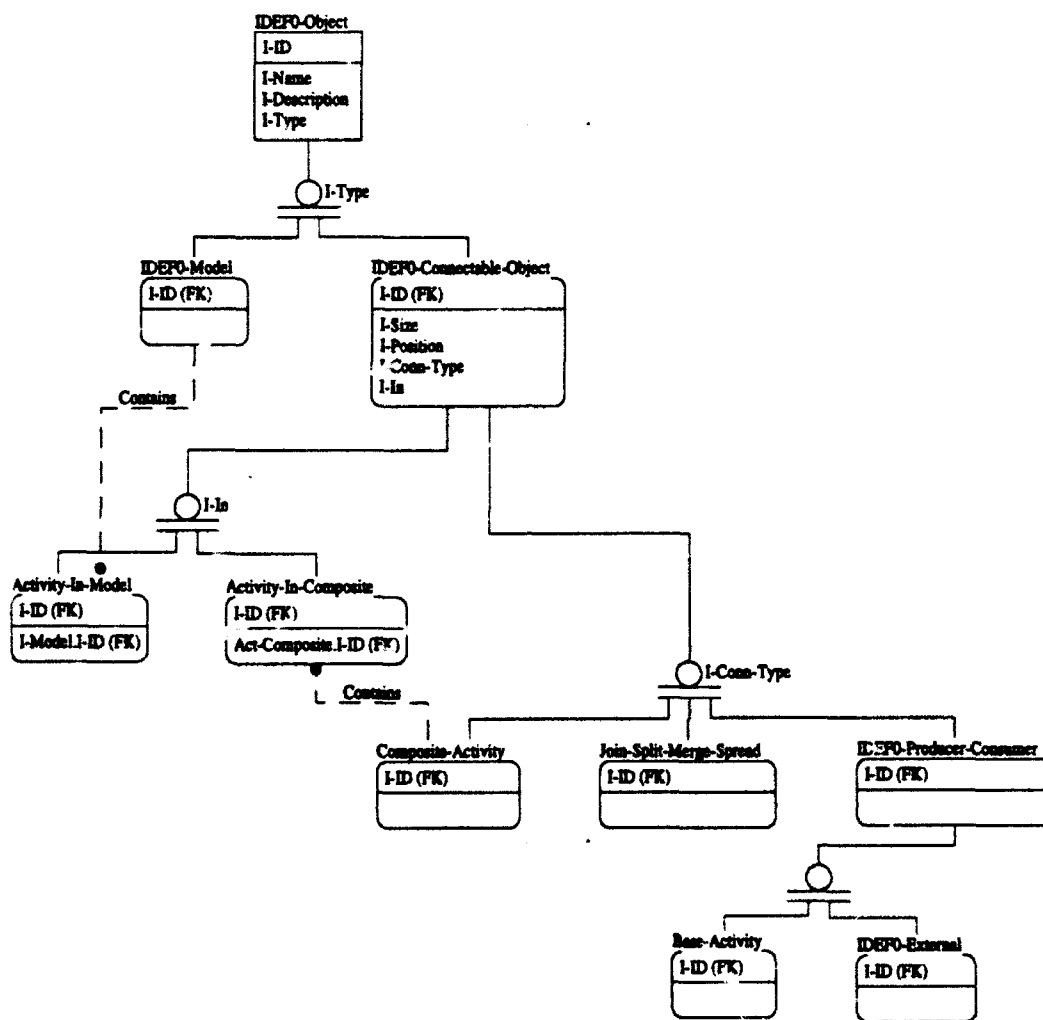


Figure 9: The IDEF0 Object Model

5.2.2 Concept Hierarchy

In the IDEF0 modeling paradigm, concepts form a hierarchal system. The entities and the relations that describe the concept hierarchy follow the generic hierarchal system developed at the beginning of this section. Every concept has a name and description stored in the attributes C-Name and C-Description. Figure 10 shows the concept hierarchy.

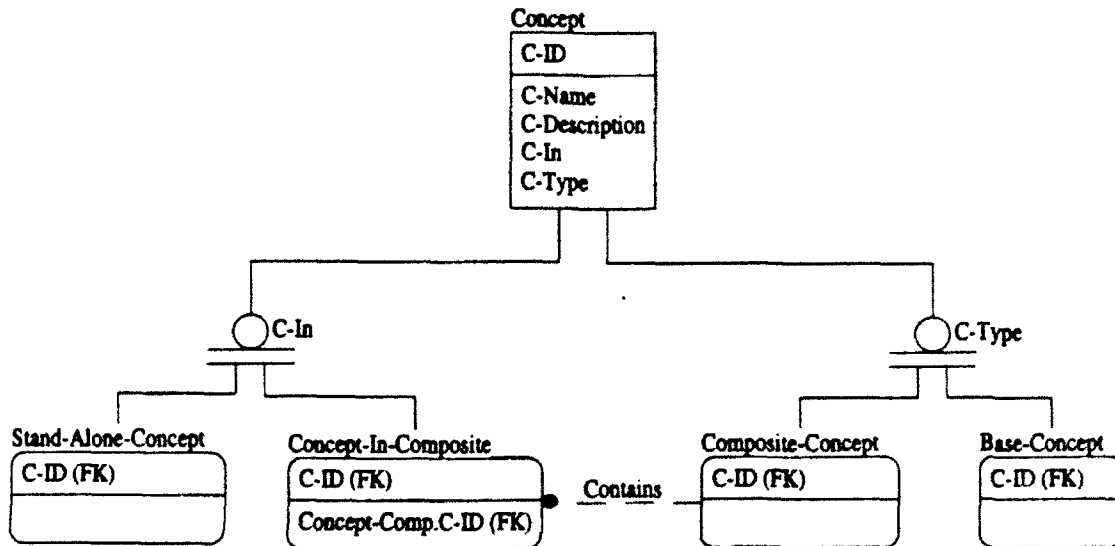


Figure 10: The Concept Hierarchal System

5.2.3 Links and Paths

Links define how a concept connects two IDEF0-Objects. Figure 11 shows the portion of the meta-model dealing with links. Each IDEF0-Connectable-Object can have several Links. This is modeled by the Contains relation. Associated with each Link is a single concept, which is modeled by the Defines relation. The start and end of a link are classified as either an input, an output, a control or a mechanism, which is recorded in the In-ICOM and Out-ICOM attributes. These attributes are given the value N/A if the connection is on an IDEF0-External or a Join-Split-Merge-Spread entity.

Links must also maintain the geometric path that will be used to draw them. This information is stored in the Link-Geometry attribute. To ease the drawing of IDEF0 diagrams, a Contains relation between Links and IDEF0 objects is created. This relation specifies all the links that are contained in the child diagram of a particular activity.

An IDEF0-Path is defined as the connection of a single base concept between two IDEF0-Producer-Consumers. If a connection between the two IDEF0-Producer-Consumers is a composite concept then there are several paths: one for each base concept in the composite concept. Since paths are only associated with base concepts there can be no joining or splitting of paths in the model. Each instance of the IDEF0-Path entity defines a path by specifying the two IDEF0-Producer-Consumers it connects and the Base-Concept that flows along it.

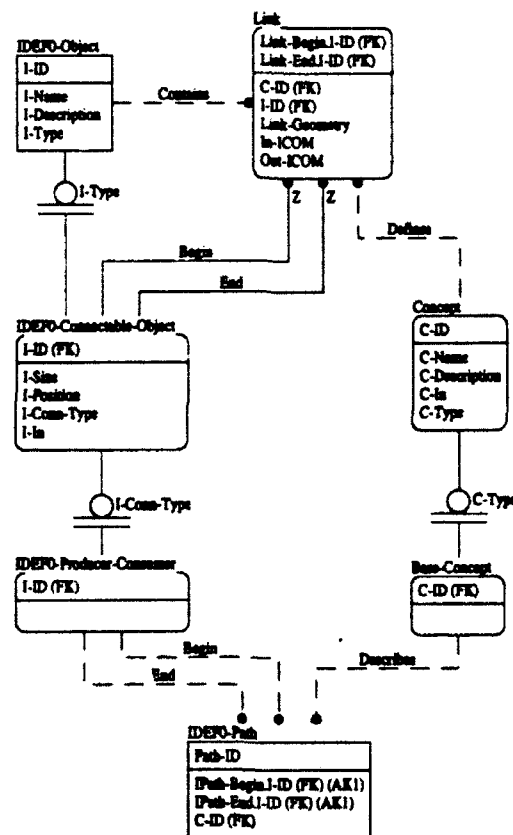


Figure 11: IDEF0 Links and Paths

5.3 XSpec Model

The XSpec meta-model is given in Figures 12, 13 and 14. A detailed description of the entities and their relations is given below.

5.3.1 Object Hierarchy

The XSpec-Object structure is very similar to the IDEF0-Object structure. Figure 12 illustrates the model for XSpec-Object entities. XSpec-Object is first classified as either a connectable object or the entire XSpec model. XSpec-Connectable-Objects follows the standard hierarchal structure introduced in the beginning of this section. Similar to the IDEF0-Object hierarchy the root objects are contained in the XSpec-Model entity. The base objects are further classified into Element and XSpec-External categories. Both of these objects are categories of the generic entity XSpec-Producer-Consumer.

Every XSpec-Producer-Consumer is an executable object and therefore is related to the entity Tool that represents the simulation tool that they are executed on. The Tool entity stores information about the simulation tool such as its name, description and the templates that are used to create the shell elements. This information is stored in the appropriate attributes of the Tool entity. In the early stages of an XSpec model, the tool which will simulate each entity may not be known. So the zero or one relation, Is-A and the Executable-Producer-Consumer entity are included so that these early XSpec models can be represented. In a completely specified XSpec model each XSpec-Producer-Consumer must also be an Executable-Producer-Consumer.

All XSpec-Objects have names and descriptions which are stored in the attributes X-Name and X-Description. The position and size of all XSpec-Connectable-Objects must be known in order to draw them. This information is stored in the attributes X-Position and X-Size.

5.3.2 Terminal Hierarchy

The terminal structure is illustrated in Fig 13. Terminal entities represent both connectors and pins in the XSpec model. Since connectors can be in a hierarchy the standard hierarchy structure is used to model XSpec terminals.

All terminals are related to the object that they are on. Terminals have a name, a description, a size, and a position attribute. The base terminal entity is a pin. Pins are further categorized into Unspecified-Pin and Specified-Pin categories. This is included in the meta-model to facilitate storing partial XSpec models. In a complete XSpec model, all pins must be fully specified.

To fully specify a pin, its Pin-Direction (whether it is an input or an output pin) and a complete definition of the message that is associated with the pin must be defined.

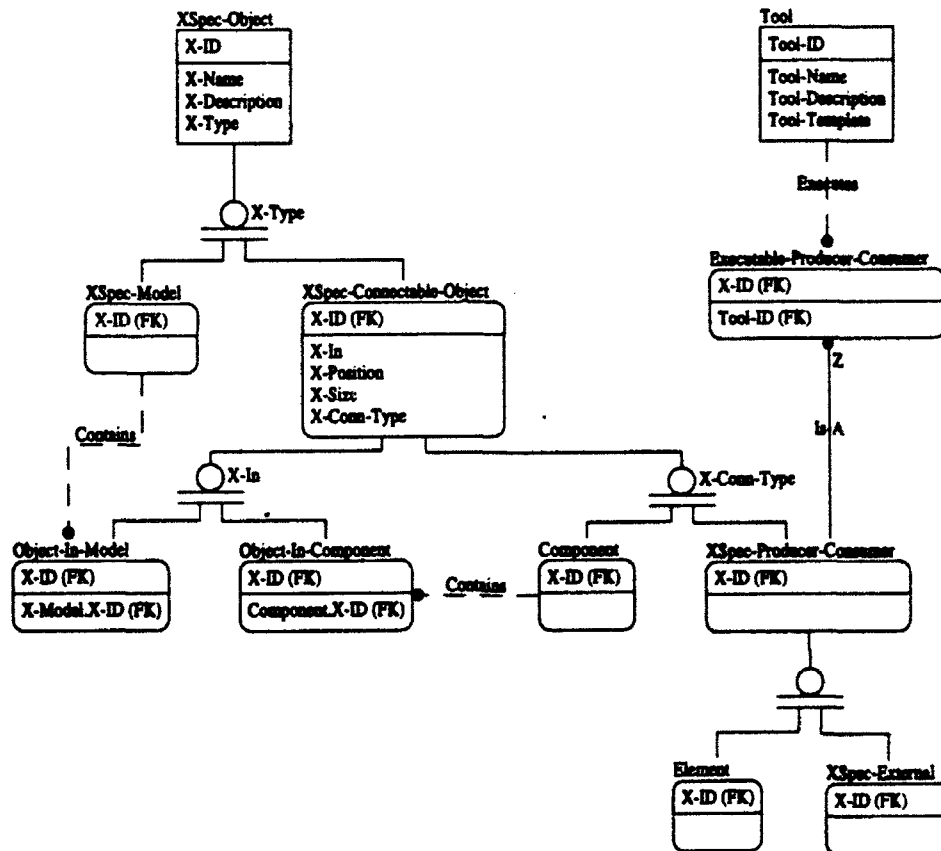


Figure 12: THE XSpec Object Model

The message definitions are specified in another entity called Message-Description and its related entities. This was done since many pins may share the same message definition. The name and description of the message are recorded in appropriate attributes of the Message-Description entity. The kind of message is recorded in the attribute M-Kind (event, signal, command, or request). Since messages contain an indeterminate number of parameters a Message-Parameter and Reply-Parameter entity are defined, which in turn are related to an Attribute entity which specifies the data type of the parameter. The Reply-Parameter is only used for request/reply message types. The Attribute entity will form a link to the IDEF1x meta-model.

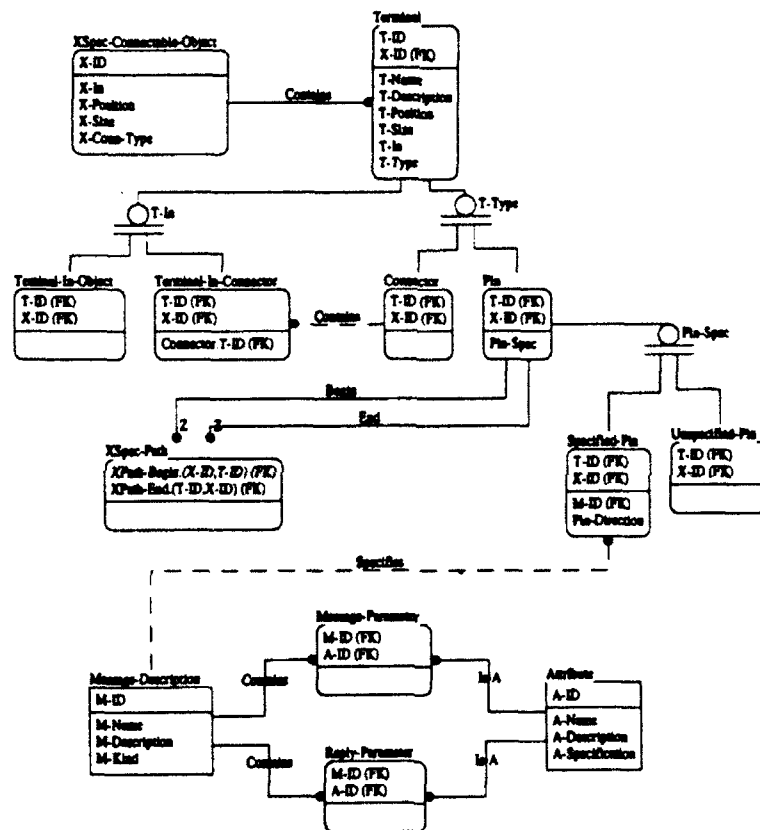


Figure 13: The Terminal Structure

5.3.3 Paths

Ultimately a pin on an XSpec-Producer-Consumer must be connected to another pin on a different XSpec-Producer-Consumer. This connection is modeled by an XSpec-Path entity (see Figure 13). XSpec-Paths play the same role as IDEF0-Paths. XSpec-Paths only need to be related to the Pins at its endpoints, since the Pin attributes contain the X-ID of the object they are on and a relation to the Message-Description. There are a few constraints on the Pins that are the terminal points of an XSpec-Path:

1. XSpec-Paths can only terminate on fully specified Pins.
2. The two Pins and their associated Message Descriptions must be compatible, that is, they must be of the same type and have the same message specification.
3. The Pin associated with an XSpec-Path through the Begin relation must have an out direction.
4. The Pin associated with an XSpec-Path through the End relation must have an in direction.

A Z cardinality are used on the Begin and End relations, since all pins do not need to be linked to a path. This allows partial models and potentially unconnected pins to be stored.

5.3.4 Cable Hierarchy

Cables join pins and connectors. They have a similar role as Links in the IDEF0 meta-model. However, since the Terminal entities form a hierarchal system, cables must also form a hierarchal system. They are modeled by the standard entity structure introduced at the beginning of this section (see Figure 14). The base cable object is a Wire. Wires are used to connect Pins. The composite object for cables is Bundle. Bundles are used to join connectors. Wires are related to the pins they connect and bundles are related to the connectors they join.

Pins and connectors on elements will have only one wire or bundle attached to them. Pins and connectors on components will have two cables attached to them. The Begin and End relation between Bundle and Connector or between Wire and Pin does not imply a direction flow of the messages bundles and wires carry.

A cable is contained in the XSpec-Object that represents the diagram that the cable is drawn in. This is modeled by the Contains relation. By the construction rules of an XSpec diagram, it is clear that a cable can be contained in only one XSpec-Object. Only the root cables (cables that are not contained in bundles) are drawn in the XSpec diagram. Therefore, only the root cables need to have an attribute to record their geometric path in the diagram. This is done by the Cable-Geometry attribute.

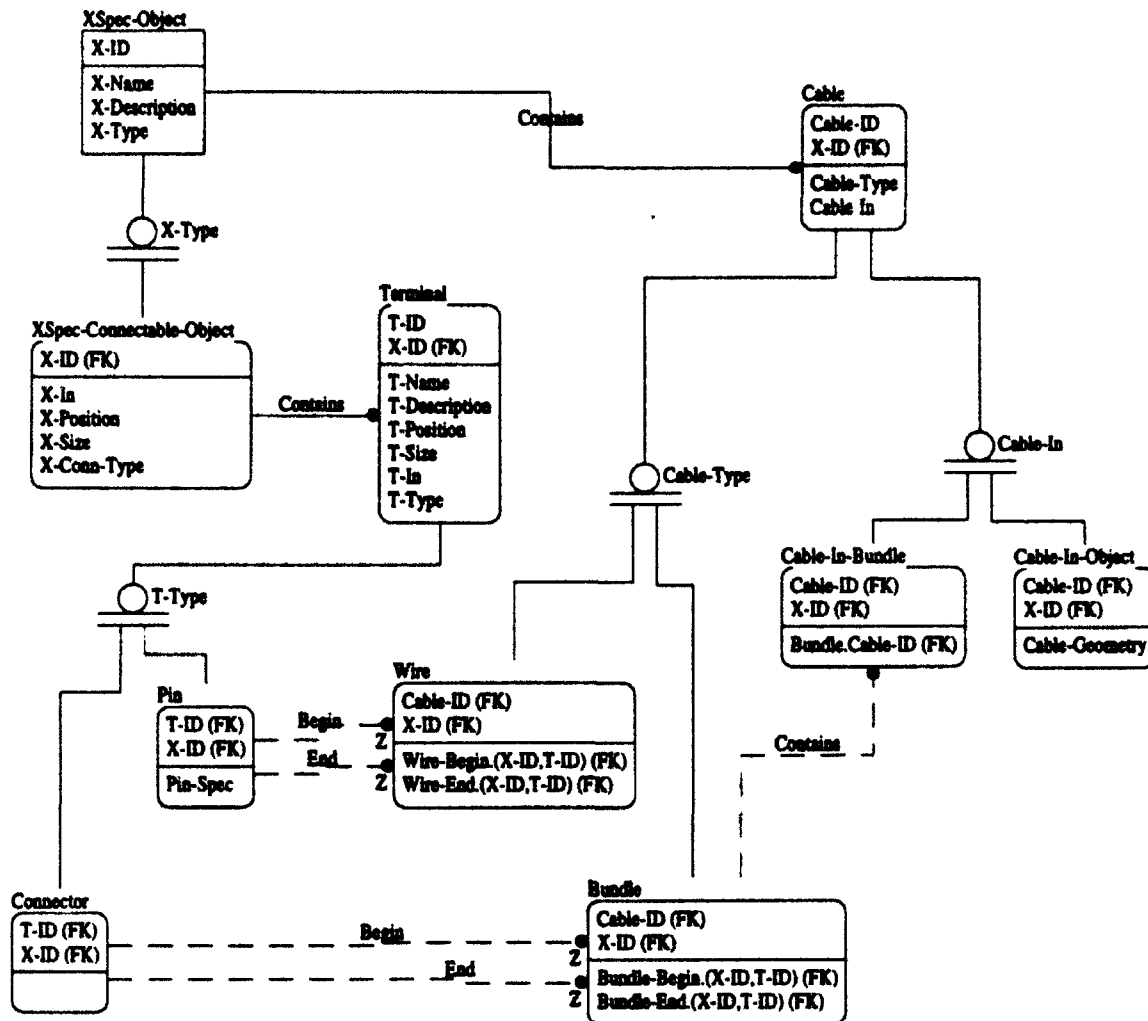


Figure 14: The Cable Hierarchy

6. The Combined Model

The objective of this research is to integrate IDEF0, XSpec, and IDEF1x models. This is accomplished by combining all three of these modeling languages into a single meta-model which describes how these languages (and models developed with these languages) are related. With this meta-model the system model can be thought of as a single representation which exists in some multi-dimensional space. IDEF0, IDEF1x, and XSpec can be viewed as modeling languages which can illustrate and manipulate a certain subset of all the dimensions of the system model.

What makes the problem difficult is that these dimensions overlap. By creating a single meta-model that represents the system, any changes that one modeling language makes are changes to the system. Then when any other language is used to view the system it should automatically view it with the changes in place. This process is analogous to modern CAD systems, where a change in a front view of a part is automatically reflected in the side and top views. This is because the change is made to the representation of the part not just to the view of the part.

This section describes the relation between the three design models. It specifies the changes that are needed in each of the design methods to support the integration. Next, the consistency rules are defined. A proposed design process using the integrated tools is given. Finally, the IDEF1x meta model of IDEF0 and XSpec is described.

6.1 Relation Between IDEF0 Model and an XSpec Model

An IDEF0 model is a high level model specifying the functions and flows among the functions. The IDEF0 model is based on a functional decomposition of the system. An XSpec model is a specification of an executable model of the system. It is much more detailed than an IDEF0 model. XSpec models specify the executable modules of the system and the flows among the modules. These modules may be either software or hardware components. These modules are decomposed in a hierarchical fashion.

In general the functional decomposition in IDEF0 and the modular decomposition in XSpec follow very dissimilar paths. What we try to accomplish in the combined model is to insure consistency at a given intermediate level in the XSpec hierarchy and the lowest level of the IDEF0 hierarchy. This allows XSpec models to add detail to the description of the system without destroying the relation between the two models.

An XSpec model is connected to an IDEF0 model through four entities: XSpec-Connectable-Object, XSpec-External, Transaction, and Foreign-Wire. These relations are explained in detail below.

6.1.1 XSpec and IDEF0 External

In an IDEF0 model, there are points which represent the generation or removal of a concept from the context of the IDEF0 model. These points are termed *IDEF0 externals*. Likewise, XSpec

models have *XSpec externals* which generate or sink messages. Any XSpec external can be associated with an IDEF0 external when their messages and concepts are related (see Section 6.1.3). Every IDEF0 external must be associated with a single XSpec external. However, an XSpec external may be associated with several IDEF0 externals. Since, XSpec models are more detailed there may be an XSpec external which is not an IDEF0 external.

6.1.2 Base Mechanisms

A *base mechanism* is an XSpec component or an element. *Base objects* are either a base external or a base mechanism. Base objects must be part of a set which is both disjoint and covers the component hierarchy tree. A set of nodes N in a tree are *disjoint* if all the paths from the leaves of the tree to the root pass through at most one node in N . The set N *covers* a tree if all the paths from the leaves of the tree to the root pass through at least one node in N . The set of nodes marked by a "+" in Figure 15 cover the tree and are disjoint.

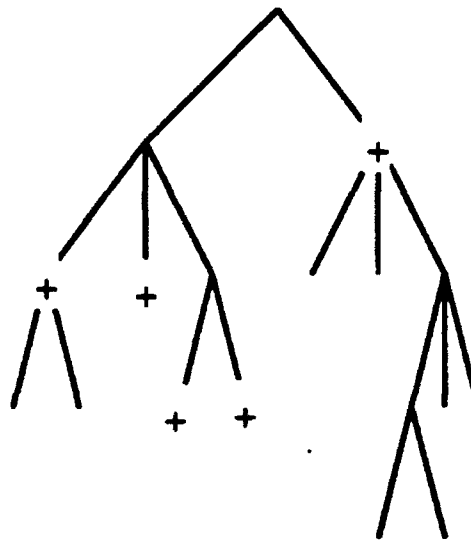


Figure 15: Example of Disjoint Nodes Which Cover a Tree

A *base activity* in an IDEF0 model is an activity which has no children. That is, the base activities in an IDEF0 model are the leaves of the activity hierarchy tree. Clearly the set of all base activities are disjoint and must cover the activity tree.

Each base activity in an IDEF0 model has a single XSpec base mechanism that has the responsibility of performing that activity. In addition every base mechanism must be responsible

for performing one or more base activities. The XSpec model represented by the set of all base mechanism is called the base *XSpec model*. Since base mechanisms are disjoint and cover the model, the base XSpec model is a complete model of the system.

Allowing only a single mechanism for each base activity is not a large restriction. When it appears that more than one base mechanism is responsible for an activity, then there is a *leveling problem* between the IDEF0 model and the base XSpec model. In this case the base XSpec model is more detailed than the IDEF0 model. When this happens, the designer should group the base mechanisms that are associated with the same base activity into a component. This component then becomes the new base mechanism. This will form a set of fewer, less detailed, base mechanisms and will balance the level of detail between the models. Another type of leveling problem occurs when a large number of base activities are mapped to the same base mechanism. In this case the IDEF0 diagram is more detailed than the base XSpec model and the designer should consider replacing the base mechanisms with a set of more detailed base mechanisms.

The mechanism notation of IDEF0 models provides a convenient way to specify base mechanisms. In addition, the portion of the XSpec component hierarchy above the base mechanisms becomes the IDEF0 mechanism hierarchy. If desired, the XSpec component hierarchy above the base mechanisms can be defined on the IDEF0 diagram through appropriate joins of the mechanism concepts.

XSpec objects are static features of the model. Therefore, base mechanisms which are XSpec components and elements must not be an output of some activity. This is not a big restriction. When IDEF0 models have an activity which is the source of a mechanism, this activity is modeling the allocation of a resource. The allocation process can alternatively be represented by a control flow which is used to select the proper resource.

6.1.3 Foreign Flows and Transactions

IDEF0 models also contain a hierarchy of concepts. These concepts can be inputs, controls, and outputs (mechanisms are now XSpec components and elements and will no longer be considered as concepts.) The concepts which are composed of other concepts are called *composite concepts* while those which have no children are called *base concepts*.

An *IDEF0 flow* is the flow of a base concept between two base activities or between a base activity and an IDEF0 external. Flows are denoted by the lines connecting the activities in an IDEF0 model. Every flow is associated with a line in the IDEF0 model. However, any given line may carry multiple flows if it is associated with a composite concept. A *foreign concept flow* is either between an external and a base activity, or between two base activities which are associated with different base mechanisms. A *local concept flow* is between base activities which are associated with the same base mechanisms. A base concept which is associated with at least one foreign concept flow is a *foreign concept*. A *local concept* is associated with only local concept flows.

As base mechanisms are used to map activities to the XSpec component hierarchy, so transactions are used to provide a mapping between foreign concepts and XSpec messages. A *transaction* is a group of XSpec messages which represents an IDEF0 foreign concept flow. XSpec messages associated with a transaction are called *foreign messages*.

A single base concept may be associated with different XSpec transactions. This allows a single IDEF0 concept to be represented by different sets of XSpec messages. For example, the transaction to move a part may be different if a human moves it or if it is done by a robot. This detail will probably not be expressed in the IDEF0 model but will require a different set of messages in the XSpec model. Every foreign flow in an IDEF0 model is associated with a single transaction.

Finally, we define the notion of a foreign message flow in an XSpec model. A *foreign message flow* is a message which is sent either between an XSpec external and a base mechanism, or between two base mechanisms. All other XSpec messages are *local messages*.

6.1.4 Consistency

We can now define the notion of consistency. Given an IDEF0 model, an XSpec model, a set of base externals, a set of base mechanisms and a set of transactions, the two models are consistent if, and only if, all the following properties hold:

1. For every foreign concept flow in the IDEF0 model, there exists a foreign message flow in the XSpec model for all the messages in the transaction associated with the concept flow. Further, all these messages must exist between the two base mechanisms/base externals associated with the base activities/IDEF0 externals that are connected by the concept flow.
2. Every foreign message flow must be associated with a transaction.

The first condition ensures that all the concept flows in the IDEF0 model are appropriately modeled by messages in the XSpec model. The second condition ensures that there are no extraneous messages in the XSpec model.

6.1.5 Remarks

Listed below are a set of remarks about the proposed way to combine the IDEF0 and XSpec models.

1. Since base mechanisms can have children in XSpec and concept flows can be represented by multiple messages, then an arbitrary amount of detail can be added to an XSpec model and it can still be checked for consistency with IDEF0.
2. For a given base mechanism, all the activities associated with this component can be identified. These activities can be collected and connected to form a

mini-IDEF0 model. This mini-model will form an IDEF0 specification of the requirements for the base mechanism.

3. The activity hierarchy is not represented in the XSpec model.
4. The concept hierarchy is not represented in the XSpec model.
5. The connector and cabling hierarchies are not represented in the IDEF0 model.
6. Local concepts and concept flows have no direct relation to messages in the XSpec model. These concept flows may appear in XSpec as messages, a network of elements within a base mechanism, or as some internal communication path within an element.
7. Internal messages have no counterpart in the IDEF0 model. Typically they represent detail not present in IDEF0.
8. Components, elements, and wires internal to a base mechanism have no counterpart in the IDEF0 model.

6.2 Relation Between IDEF1x and XSpec Models

XSpec models the dynamic behavior of a system. IDEF1x models the data that is used by the system. These two models do not share many features. The only link between the two is that the parameters on the messages that query the database must be attributes in the IDEF1x model.

The link is accomplished by adding a store element to XSpec. Unlike other XSpec elements, this is not a model of some dynamic portion of the system. Rather, this new type of element is a window into the database for the system model. Messages to a store element can add, delete, change, or interrogate the data stored in the database. As such all the parameters on all messages terminating on a store element must be attributes in the IDEF1x model. Further the the message parameters can be examined to see if they contain sufficient information to locate the data element of interest.

In the combined model, each XSpec parameter is classified as either a dynamic parameter or a storage parameter. All parameters on all messages terminating on a store element must be storage parameters. All storage parameters are attributes in the IDEF1x model.

None of the IDEF0 or XSpec model structures are used in the IDEF1x model. They may suggest possible entities and relations, but are not directly used. Likewise, the entity diagram structure of the IDEF1x model have no counterpart in either the IDEF0 or XSpec model.

6.3 Relation Between IDEF1x and IDEF0 Models

There has been effort by others to use an IDEF0 to help generate IDEF1x models. Most of this effort has focused on the using the concepts in IDEF0 as a start for identifying the entities and attributes for the IDEF1x model. This can still happen. It is recommended that first an XSpec model be created from the IDEF0 model and then the message parameters can become the attribute pool. An alternative method is to develop the IDEF1x model from the IDEF0 model first, and then use the attribute list as a specification for many of the messages used in XSpec. In fact, as long as the parameter list is frequently updated, the IDEF1x and XSpec modeling efforts can proceed in parallel once the IDEF0 model is finished.

6.4 Restrictions Placed on Models

This section summarizes the additions and restrictions the combined modeling method has placed on the three modeling tools.

IDEF0

1. The mechanism hierarchy becomes the XSpec component hierarchy.
2. Mechanisms are no longer part of the concept hierarchy.
3. Each base activity must be mechanized with a single base mechanism.
4. Mechanisms cannot be an output of any activity.
5. Every IDEF0 external must be associated with an XSpec external.
6. Every foreign concept must be associated with at least one transaction.

IDEF1x

1. The concept hierarchy and the transaction definitions can be used as sources for the entity pool.
2. There must be an IDEF1x attribute for each storage parameter.

XSpec

1. Base mechanisms have to be identified.
2. Base mechanisms must be a set which is both disjoint and covers the component hierarchy.

3. At least one base activity must be associated with each base mechanism.
4. XSpec externals associated with IDEF-0 externals must be identified.
5. Foreign messages must be associated with transactions.
6. There must be a foreign message flow for every message in each of the transactions associated with foreign concept flows.
7. Partial IDEF0 models are used as the requirements specification for a base mechanism.
8. Store elements are added.
9. Storage parameters must be identified.

6.5 The Modeling Process Using the Combined Toolkit

This section describes a typical modeling process using an integrated system of the three modeling languages described in this report: IDEF0, IDEF1x, and XSpec. The process assumes that after the IDEF0 model is created, an XSpec model is developed. This is good practice if there are significant dynamic elements in the system to be investigated. If the system is primarily a data intensive application, then the creation of the XSpec model could be eliminated. The modeling process consists of the following major steps.

1. **Gather information.** The first step is to gather information about the system to be modeled. Information sources usually include: documents about the system, interviews with experts, surveys of the users of the system, observation of the existing operations, and information from previous related projects.
2. **Generate an IDEF0 model.**
 - a. **Create activity and data lists.** From all the information that was gathered, list all potential concepts and group the concepts into potential aggregates. List all the functions or activities that are performed by the system. Begin clustering the activities to create an initial activity hierarchy.
 - b. **Define context.** Draw the context diagram to clearly indicate the scope of the system.
 - c. **Create an IDEF0 model of the system.** Decompose the activities in the system. Connect the activities with appropriate concepts. Refine the definitions of the activities and concepts. Do not mechanize this model. Review and refine the IDEF0 model.

3. Mechanize the IDEF0 model.

- a. **Identify the base mechanisms for the model.** A base mechanism is a physical object or software module that performs a base activity in the IDEF0 model. Group the base mechanism into a hierarchical structure. This structure is the initial component hierarchy.
- b. **Fully mechanize the base activities in the IDEF0 model.** Every base activity in the IDEF0 model must be assigned one, and only one, base mechanism. A base mechanism can perform more than one base activity.

4. Generate a prototype XSpec model. From the IDEF0 model, generate a prototype IDEF0 model. This process is fully automatic.

5. Refine the XSpec model.

- a. **Refine each base mechanism.** Depending on the complexity of the base mechanism, determine whether it should be an element or a component. Create the decomposition of the component if necessary.
- b. **Refine each transaction.** Determine the set of messages required to perform each transaction. Completely specify each message.
- c. **Continue to refine the message protocols.** Check the inter-operability of each element. Add new messages and refine the specification of others as necessary. Add new elements and component structures as necessary.
- d. **Resolve Inconsistencies.** Ideally, the combined model will keep the XSpec and IDEF0 models consistent with one another automatically. However, since the models are not edited on tools designed to interact with the combined model, consistency is checked when the the combined model is updated, a report is generated, and the modeler resolves the inconsistencies.
- e. **Review the XSpec model.** Review the final model. Make sure that each base mechanism performs all the required activities that were specified in the IDEF0 model. Refine the model as necessary.

6. Develop an IDEF1x model

- a. **Identify entities.** A good source for possible entities are transactions. These form natural groupings of messages. Determine all of the transactions that contain at least one message whose parameter is an attribute. Consider the transaction as a possible entity.

- b. **Identify attributes.** Identify the the message parameters that are potential attributes. Not all parameters may be attributes, since messages may model physical flows as well as data flows.
 - c. **Construct IDEF1x model.** Determine the relations among the entities, draw the entity model, determine key attributes, and determine the owners of the non-key attributes.
 - d. **Review and refine the model.** Review the model, check for consistency with IDEF1x syntax rules, and add refinements as necessary.
 - e. **Update the XSpec parameters.** This is mostly an automatic process since parameters are attributes. A problem that will occur if the IDEF1x model deletes an attribute which was a parameter on a message. A consistency checker will flag these, a report generated, and the modeler must adjust one of the models.
7. **Identify inconsistencies between the IDEF1x model and the XSpec model.** XSpec models all access to information in a database as queries to a store element. The parameters in these queries must be attributes in the IDEF1x model. The parameters can be checked to be sure that there is enough information to access the proper entity instances.
 8. **Generate the simulation templates.** Generate the templates for the executable models of the XSpec entities. This is an automated procedure. The database schema can be automatically generated from the IDEF1x model.
 9. **Code the elements.** Create the executable simulation code for each element template.
 10. **Simulate and refine models.** Simulate the executable model. Adjust the IDEF0, XSpec and IDEF1x models based on the performance of the simulations.

Figure 16 depicts how the various modeling tools cooperate in the development of a common model that crosses the boundaries between IDEF0, IDEF1x and XSpec. The shaded parts of the figure are items being developed to support a common model development process.

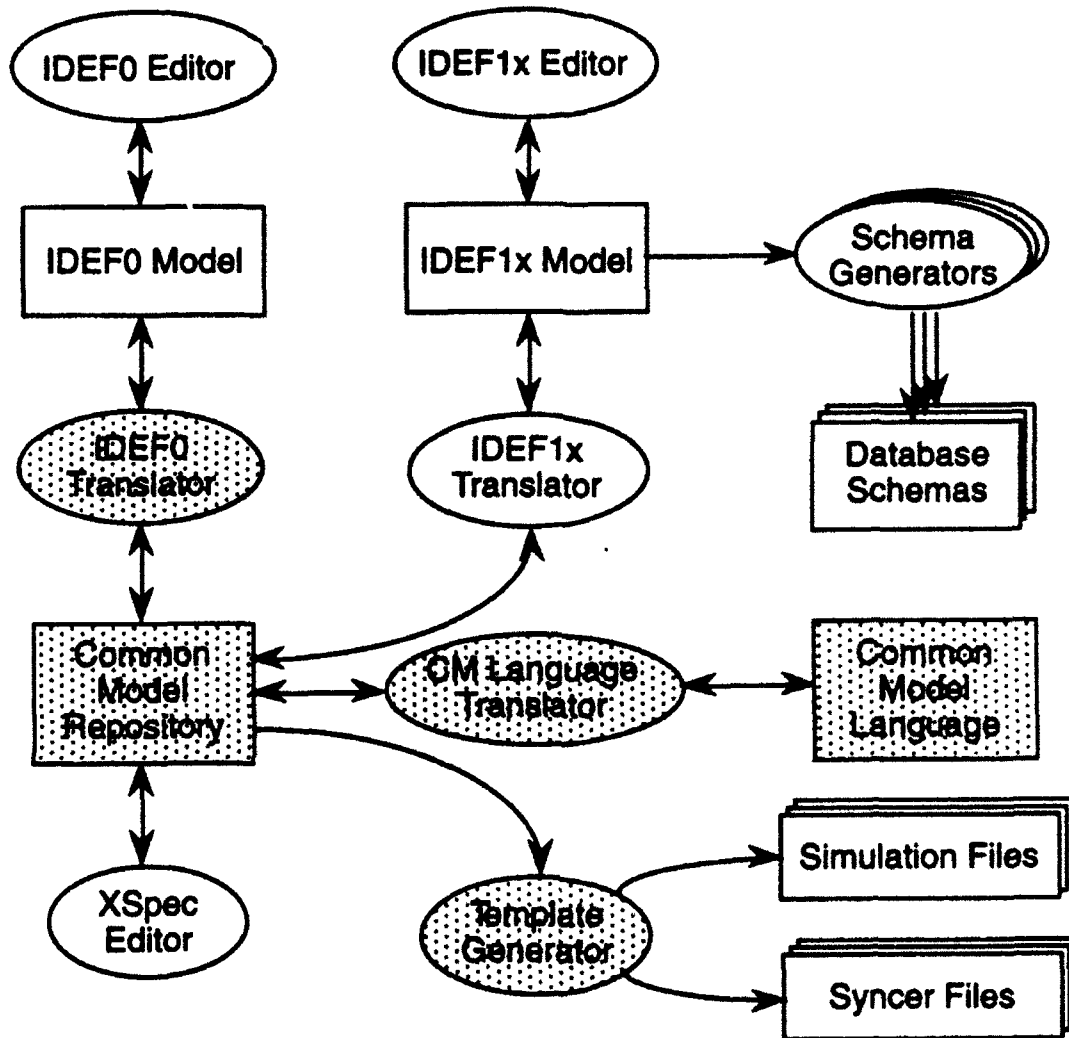


Figure 16: The Combined Toolkit

6.6 The Combined Meta-Model

This section combines the individual meta-models of the IDEF0, IDEF1x, and XSpec languages, into a single comprehensive meta-model that can be used to represent complicated systems. Much of the theory of how the models can be combined was given in the previous section. This section details how these theoretical aspects are implemented in the meta-model.

6.6.1 Base Mechanisms and Externals

The relation between XSpec-Connectable-Object and Base-Activity is used to identify which XSpec-Objects are base mechanisms (see Figure 17). This relation identifies the Base-Activities in the IDEF0 model that are associated with each base mechanism.

An XSpec-External which produces or consumes messages that correspond to a concept that is generated or consumed by an IDEF0-External is related to that external. Since XSpec is a more detailed model, not all XSpec externals need to be IDEF0 externals. Recall that in IDEF0, an external is a source (or sink) of a single concept. In XSpec an external may initiate several transactions, so the Consists-Of relation must have a 1 to many cardinality.

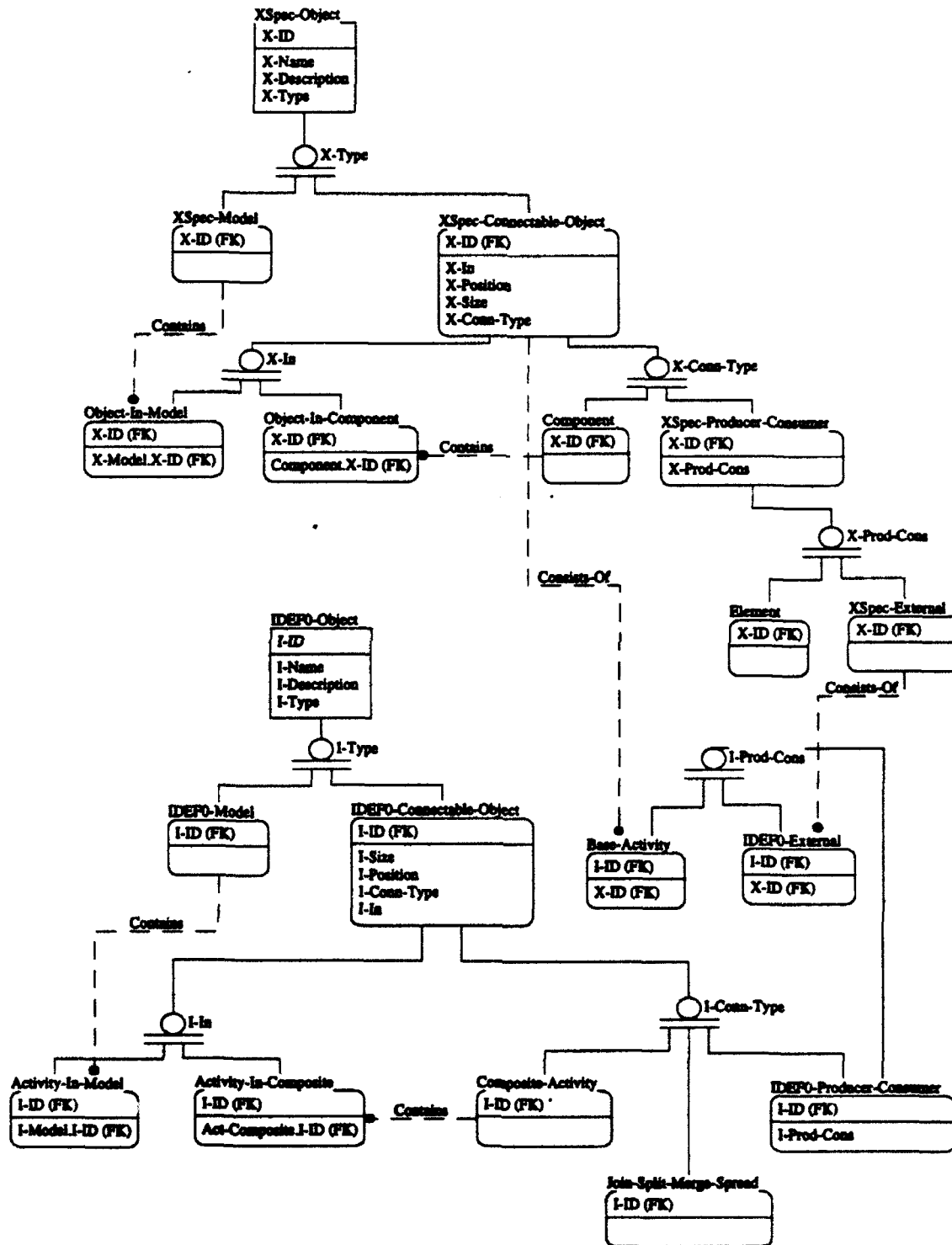


Figure 17: Base Mechanisms and Externals

6.6.2 External concepts

An IDEF0-Path is classified as either a Local-Path or a Foreign-Path. A Foreign-Path is any path that connects two base activities which are associated with different base mechanisms. The Foreign-Path entity is the major component that keeps the XSpec and IDEF0 models consistent. The Foreign-Path entity will be explained in more detail later. As soon as all the Base-Activities and IDEF0-Externals are completely defined then IDEF0-Paths can be automatically categorized as either local or foreign.

If a Base-Concept Describes an IDEF0-Path that has been categorized as a Foreign-Path, then this Base-Concept will be categorized as a Foreign-Concept. All concepts that are not Foreign-Concepts are Local-Concepts. This categorization is easy to automate.

The categorization of concepts is illustrated in Figure 18.

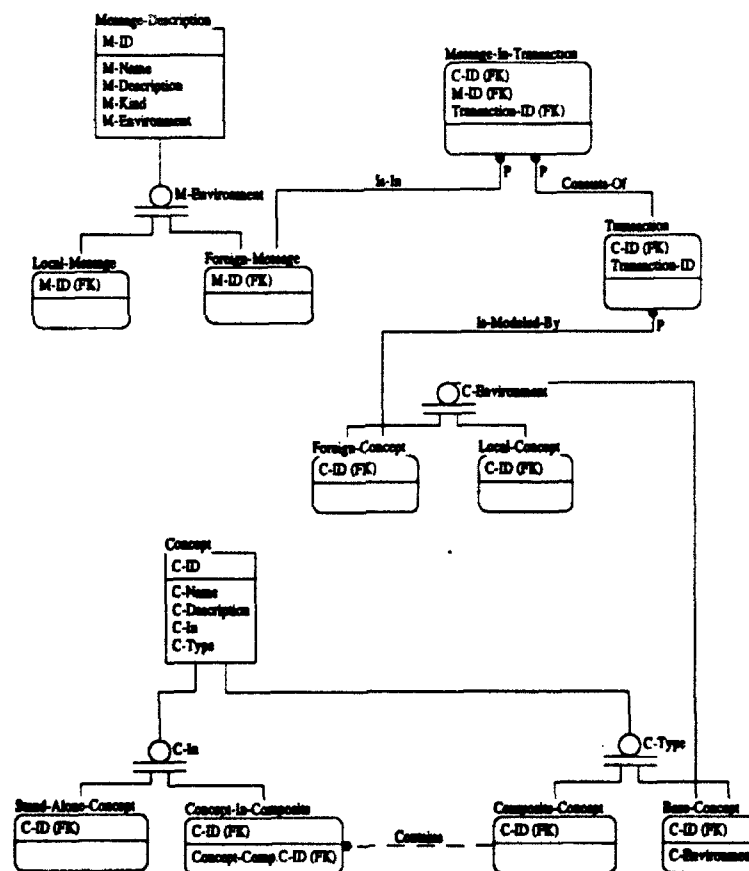


Figure 18: Relation of Messages to Concepts

6.6.3 External Messages

Message-Description entities are classified as either Local-Messages or Foreign-Messages. Foreign-Messages are those that correspond to an IDEF0 concept flow, that is a message which flows between two different base mechanisms. Messages which flow only within the same base mechanism are Local-Messages. The classification relation is shown in Figure 18.

6.6.4 Transactions

A transaction is a mapping between a concept and a set of XSpec messages. The Consists-Of relation represents this mapping. Since a message may be a part of many transactions, and a transaction consists of many messages, the entity, Message-In-Transaction is used to break up the many-to-many relation. Occasionally, the same IDEF0 Concept has to be modeled as several different protocols in an XSpec model. This is due to the added detail in the XSpec model. The protocol of messages needed to transfer a part may depend on the base mechanisms doing the transfer. For example the protocol would be different if the transfer was done by a human than by a machine. This situation is modeled by the 1 to many Is-Modeled-By relation between the Foreign-Concept and Transaction entities.

6.6.5 Foreign Path

Every foreign path is related to a set of links which connect two base activities of IDEF0 Externals. A single-link (which represents the flow of a composite concept) can be associated with many paths. The entity, Link-Associated-With-Foreign-Path, is used to break up this many-to-many relation (see Figure 19). A Foreign-Path also has a unique Transaction associated with it. This is determined by the Flows-Along relation. Relations also exist between a Foreign-Path and the Foreign-Wires which constitute the path. This is done for convenience, so that if any of these paths or wires are edited, the modeling tool knows that the models are no longer consistent and the appropriate actions can be taken.

Notice that wires are classified into two categories. Those related to Foreign-Paths, and those that are not related. All wires that connect base mechanisms together, must be related to a Foreign-Path. These are classified as Foreign-Wires. The wires that exist internal to a base mechanism decomposition are not related to the IDEF0 model and are classified as Local-Wires. The Foreign-Path entity and all of its relations are shown in Figure 19.

Notice that Foreign-Paths are not related to XSpec-Paths. Only the portion of each XSpec-Path that flows between base mechanisms must be consistent with a Foreign-Path in the IDEF0 model. Changes to the XSpec-Path internal to a base mechanism do not impact consistency. Since XSpec-Paths are identified only by their terminal points, a relation to XSpec-Paths and IDEF0-Paths is not desired.

The entire combined IDEF0 and XSpec meta-model is shown in Figure 20.

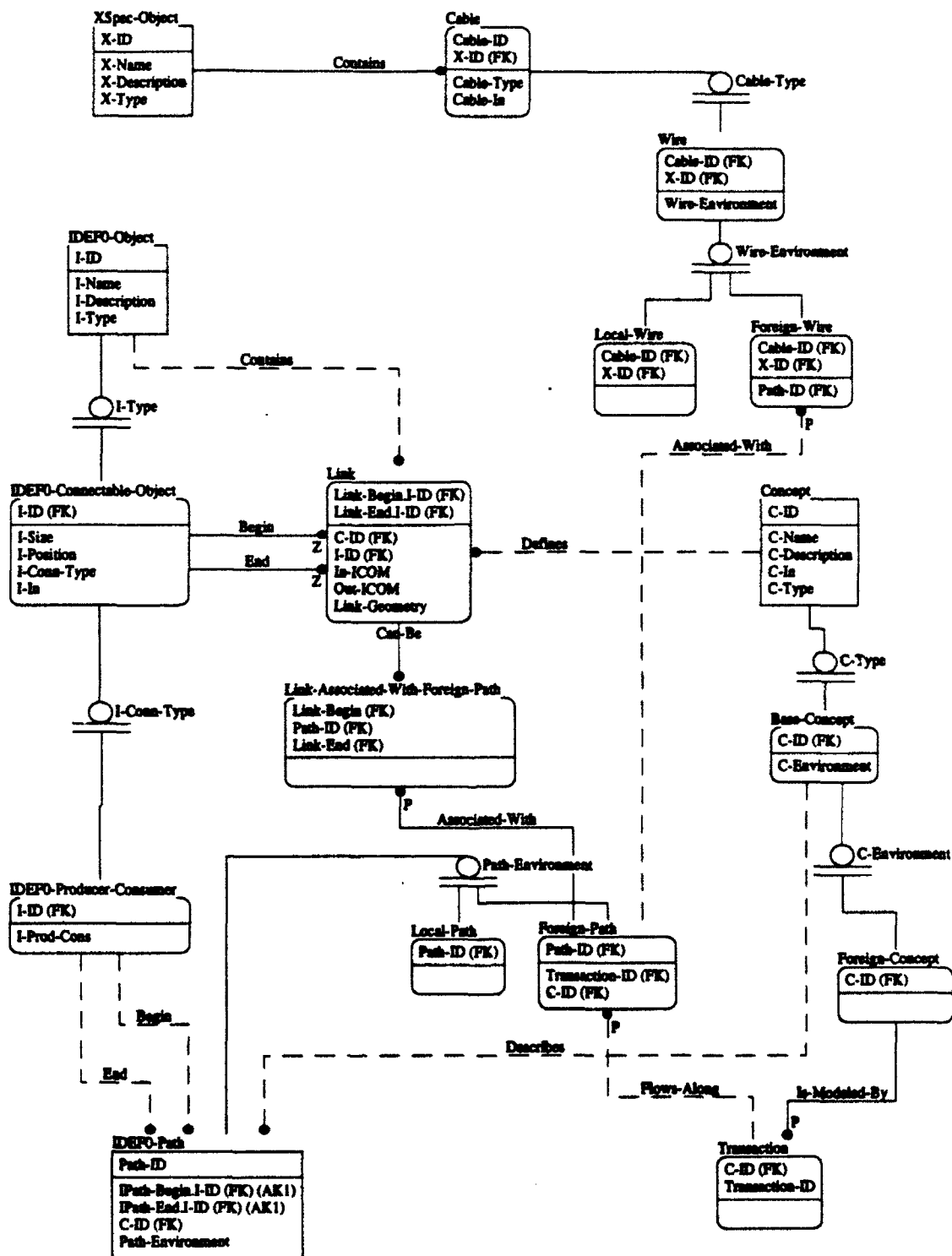
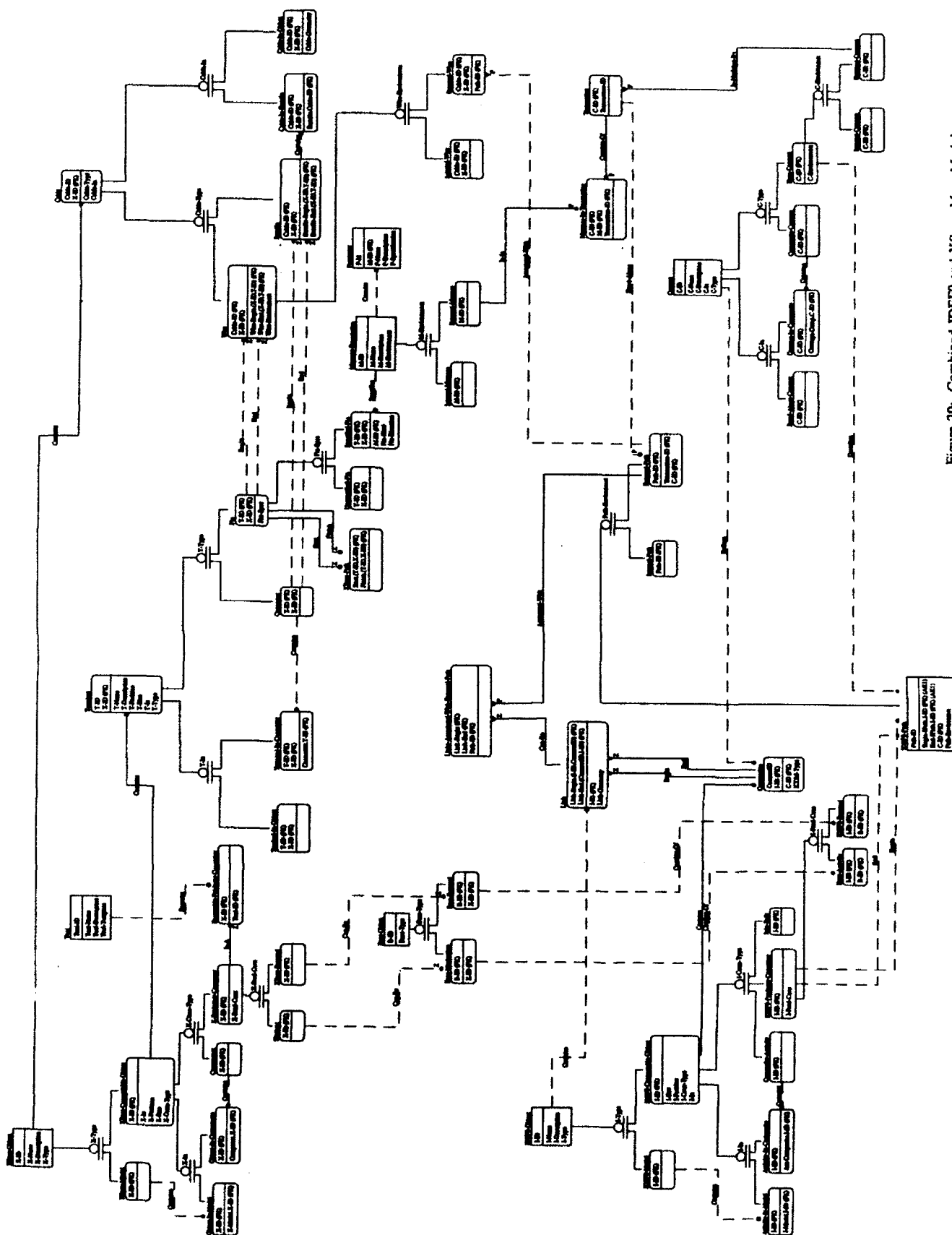


Figure 19: Relation of Paths and Wires



6.7 Generating a Prototype XSpec Model

This section describes how a prototype XSpec model can be generated from an IDEF0 model. Ideally, as the IDEF0 model is being developed, the XSpec data elements would be automatically generated and stored in the combined model database. Then at any point in the IDEF0 model development an XSpec model would exist. However, since we are integrating commercial tools, they are not designed to interact with our combined model, and so another approach will be taken. After the IDEF0 model is completed, the designer will execute a translation program. This program will translate the output of the IDEF0 modeling tool into the combined model. It then generates a prototype XSpec model and fills in the XSpec entities. This section describes an algorithm to generate the prototype XSpec model.

6.7.1 Defaults

XSpec models are more detailed than IDEF0 models. Two approaches can be taken to add the detail when generating the prototype model. First, the user could be asked to fill in the added information interactively, or second, reasonable defaults could be assigned, and the modeler fills in the detail while using the XSpec tool. The latter approach was adopted for this project since it was felt that adding detail is really part of the design process and should be done in a tool developed for that purpose. Listed below are all the default rules in the generator that will be written for this project.

1. The mechanism hierarchy will be used to generate the component hierarchy.
2. All base mechanisms will initially be modeled as components with no internal structure. This will allow further decomposition.
3. An XSpec external is created for every IDEF0 external and placed in the XSpec context diagram.
4. All foreign concepts will have one transaction. This transaction will have one message. The message name and description will be the same as the name and description of the associated foreign concept. Therefore, in the prototype model there is a one-to-one mapping between an XSpec message and an IDEF0 foreign path.
5. Messages will have no parameters associated with them.
6. Pins on base mechanisms and components will be assigned the name of the message that flows through it. If there is a duplication of a pin name in a connector, a unique suffix is added.
7. The messages will have a default M-Kind of event. The direction of the pins will be consistent with the associated foreign path.

8. Pins will be grouped into connectors based on common destinations. The names of the connectors will be the name of destination. Destinations are defined as the next XSpec object along the message path that is at the same or higher level in the component hierarchy. There will be no subconnectors.
9. Bundles will connect all the connectors in an XSpec diagram. These bundles will contain only wires. There will be no sub-bundles.

6.7.2 Generation Algorithm

Figure 21 describes an algorithm that will generate a prototype XSpec model from an IDEF0 model. The heart of the algorithm is the recursive routine CreateXSpecDiagram. This algorithm is described in Figure 22. The prototype generator expects as inputs a complete IDEF0 model, the base mechanism hierarchy, and all base activities assigned one base mechanism.

Create XSpec prototype model

- 1) Copy the base mechanism hierarchy into the XSpec component hierarchy.
- 2) Create an XSpec-External for each IDEF0-External. Place this external at the XSpec-Model level in the component hierarchy.
- 3) For each IDEF0-Path
 - a) Classify it as a local or foreign path
 - b) If it is foreign create the necessary Link-Associated-With-Foreign-Path and Link
- 4) For each Base-Concept
 - a) Classify it as local or foreign
 - b) If it is foreign create a transaction for each path related to the concept.
- 5) For each Transaction
 - a) Link the Transaction to the appropriate Foreign-Path.
 - b) Create a Message-Description for the Transaction.
- 6) Execute CreateXSpecDiagram (XSpec-Model)
- 7) Link all the Foreign-Wires to the appropriate Foreign-Path.

Figure 21: Prototype Model Generator

CreateXSpecDiagram(Object)

- 1) If Object is a base mechanism
 - a) For every foreign path on every base activity on the Object
 - i) Create a pin
 - ii) Link pin to appropriate Message-Description
 - b) Group pins into connectors by common destination.
- 2) Otherwise (We need to create an XSpec diagram for the Object)
 - a) For every XSpec-Object in Object
 - i) CreateXSpecDiagram(XSpec-Object)
 - b) For every pin on every XSpec-Object in Object whose destination is outside of Object
 - i) Create a pin on Object
 - c) Group pins into connectors by common destination.
 - d) Create the bundles that connect all the connectors in the diagram.
 - e) Create the wires in the bundles.
- 3) Return

Fig 22: Recursive XSpec Diagram Generator